PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ELIÃ RAFAEL DE LIMA BATISTA

ENHANCING EARLY SCHEDULING IN PARALLEL STATE MACHINE REPLICATION

Porto Alegre
2020

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ENHANCING EARLY SCHEDULING IN PARALLEL STATE MACHINE REPLICATION

ELIÃ RAFAEL DE LIMA BATISTA

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Fernando Luis Dotti
Co-Advisor: Prof. Fernando Pedone

Porto Alegre
2020

Ficha Catalográfica

B333e Batista, Eliã Rafael de Lima

Enhancing Early Scheduling in Parallel State Machine Replication / Eliã Rafael de Lima Batista . – 2020.
80 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Luis Dotti.
Co-orientador: Prof. Dr. Fernando Pedone.

1. Early Scheduling. 2. State Machine Replication. 3. Work Stealing. I. Dotti, Fernando Luis. II. Pedone, Fernando. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Eliã Rafael de Lima Batista

Enhancing Early Scheduling in Parallel State Machine Replication

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifical Catholic University of Rio Grande do Sul.

Sanctioned on March 18, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Andrey Elisio Monteiro Brito (Universidade Federal de Campina Grande)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS)

Prof. Dr. Fernando Luis Dotti (PPGCC/PUCRS - Advisor)

Prof. Dr. Fernando Pedone (University of Lugano/Switzerland - Co-Advisor)

“ Have you ever traveled to where snow is
made, seen the vault where hail is stockpiled,
The arsenals of hail and snow that I keep
in readiness for times of trouble
and battle and war?
Can you find your way to where lightning is
launched, or to the place
from which the wind blows?
Who do you suppose carves canyons for the
downpours of rain, and charts
the route of thunderstorms ?
That bring water to unvisited fields, deserts
no one ever lays eyes on,
Drenching the useless wastelands so they’re
carpeted with wildflowers and grass?
And who do you think is the father
of rain and dew, the mother
of ice and frost?
You don’t for a minute imagine
these marvels just happen, do you? ”

(*The Message, Job 38, 22-30*)

ACKNOWLEDGMENTS

First and foremost, praises and thanks to God, the Creator, for His unconditional blessings upon me.

I would like to express my deep gratitude to my wife Larissa, for her support, comprehension and continuous encouragement throughout such intense two years of my master's degree. She was, as always, essential and has done the impossible to allow me to focus on my studies and to carry out this work. Also to my parents, who guided me wisely to the path of Truth, where I still ride today. For their love, prayers, caring, and sacrifices for educating and preparing me for my future. To my family and friends, which have shown great support.

I am extremely grateful to my advisor Prof. F. Dotti, for giving me the opportunity to collaborate and for providing invaluable guidance throughout this research. His vision, sincerity and motivation have deeply inspired me. He taught me the methodology to carry out the research and to present the results as clearly as possible. It was a great privilege and honor to work and study under his guidance.

I would also like to thank my co-advisor Prof. F. Pedone, for the continuous support and related research, for his patience, motivation, and immense knowledge. His guidance and collaboration were essential to my research and writing of this dissertation. I extend this gratitude to Prof. E. Alchieri, who also contributed with valuable knowledge and advice, and to Prof. L. Gustavo, who evaluated my researching plan and progress since the beginning, providing important feedback.

I also want to thank the Pontifical Catholic University of Rio Grande do Sul and its faculty for its commitment to the quality and excellence of teaching.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

APRIMORANDO EARLY SCHEDULING EM REPLICAÇÃO MÁQUINA DE ESTADOS

RESUMO

Replicação Máquina de Estados é uma abordagem bem difundida cujo objetivo é fornecer tolerância a falhas em sistemas distribuídos. Baseia-se na ideia de que réplicas devem executar operações de forma determinística e, portanto, sequencial. No entanto, para se beneficiar de arquiteturas multi-core, técnicas têm sido propostas para permitir execução paralela. Elas se baseiam no fato de que operações independentes (que não compartilham nem alteram estado compartilhado) podem executar em paralelo.

Early Scheduling (escalonamento antecipado) é uma técnica promissora que abre mão de concorrência em troca de rápidas decisões de escalonamento. Nela, requisições são agrupadas em classes, e um sub-conjunto de threads é atribuído a cada classe, respeitando dependências entre requisições. Early Scheduling apresentou ganhos significativos de performance em replicação máquina de estados, apesar das restrições impostas pelas definições de classes e mapeamentos para threads.

Neste trabalho avaliamos os impactos causados por tais restrições. A partir dos resultados observados, propomos e implementamos melhorias, apresentamos os resultados e comparamos com outras abordagens clássicas. Nossa contribuição se dá no uso de técnicas de sincronização alternativas, como adaptações de espera ocupada e aplicação de conceitos de work-stealing (roubo de trabalho) ao algoritmo original.

Palavras-Chave: Escalonamento antecipado. Replicação Máquina de Estados. Roubo de trabalho. Espera ocupada.

ENHANCING EARLY SCHEDULING IN PARALLEL STATE MACHINE REPLICATION

ABSTRACT

State machine replication is a well-established approach to provide fault tolerance. One of its key assumptions is that replicas must execute operations deterministically and thus serially. However, to benefit from multi-core servers, some techniques were built to allow concurrent execution of operations in state machine replication. Invariably, such techniques are based on the fact that independent operations (which do not share common state or do not update shared state) can execute in parallel.

Early Scheduling is a promising technique that trades concurrency for expeditious scheduling decisions. In this technique, requests are grouped in classes and a fixed subset of threads is assigned to each class, respecting request dependencies. Early scheduling has been shown to provide significant performance improvements in state machine replication. Although early scheduling shows performance gains, it restricts concurrency according to the class definitions and class-to-threads mapping.

We evaluate the impact of these restrictions imposed by the early scheduling technique. Out of the observations, we propose and implement improvements to the basic technique, present their results and compare the resulting system to more classic approaches. Our main contribution concern the use of alternative synchronization techniques, such as busy-wait adaptations and application of work-stealing techniques within the former Early Scheduling algorithm.

Keywords: Early Scheduling. State Machine Replication. Work-Stealing. Busy-Wait.

LIST OF FIGURES

Figure 4.1 – Request classes definition with 2 shards. 36
Figure 5.1 – Request classes definition with 4 and 8 shards 42
Figure 5.2 – Metrics and throughput in 10 seconds (left) and an entire execution (right), balanced workload, low conflicts, 2 shards (4 threads), and light operations. 46
Figure 5.3 – Results for 2 shards, 4 threads, light costs, with balanced workloads (top) and skewed workloads (bottom). 47
Figure 5.4 – Results for 8 shards, 16 threads, light costs, with balanced workloads (left) and skewed workloads (right). 48
Figure 5.5 – Results for 4 shards, 8 threads, different operation costs, with balanced workloads. 49
Figure 7.1 – Results for single shard, light costs (top) and moderate costs (bottom). 68
Figure 7.2 – Work-stealing algorithms comparison: single shard, light costs (top) and moderate costs (bottom). 70
Figure 7.3 – Results for single shard, balanced workloads, light costs (top) and moderate costs (bottom). 71
Figure 7.4 – Results for 1% global requests, balanced workload (top) and skewed workload (bottom). 72
Figure 7.5 – Results for 15% global requests, balanced workload (top) and skewed workload (bottom). 73

LIST OF TABLES

Table 4.1 – A possible mapping of 4 threads in Figure 4.1 37
Table 5.1 – Threads to classes mappings for 4 shards and 8 threads 43
Table 5.2 – Threads to classes mappings for 8 shards and 16 threads 43

LIST OF ALGORITHMS

1 General Early Scheduling Definitions used in Algorithms. 38
2 Early Scheduler. 38
3 Worker Threads for Early Scheduling. 39
4 General Busy-Wait Early Scheduling Definitions. 51
5 Worker Threads For Busy-Wait Early Scheduling. 52
6 General Work-Stealing Definitions. 55
7 Work-Stealing Scheduler. 55
8 Work-Stealing Algorithm For Each Worker Thread *t*. 56
9 General Semi-Blocked Work-Stealing Definitions. 59
10 Scheduler For Stealing while Synchronizing. 59
11 Worker Thread *t* for Semi-Blocked Work-Stealing 60
12 General Barrier-free Work-Stealing Definitions. 62
13 Worker Thread for Barrier-free Synchronization 63
14 The Steal Procedures for Optimistic Work-Stealing 65

CONTENTS

1 INTRODUCTION 23
1.1 CONTRIBUTIONS 24
1.2 ORGANIZATION 25

2 BACKGROUND 27
2.1 SYSTEM MODEL 27
2.2 CONSISTENCY 27
2.3 REQUEST INDEPENDENCE 28

3 RELATED WORK 29
3.1 SCHEDULING 29
3.2 STATE MACHINE REPLICATION 30
3.3 WORK-STEALING 31

4 EARLY SCHEDULING 35
4.1 REQUEST CLASSES 35
4.2 CLASSES, THREADS AND EXECUTION MODEL 36
4.2.1 EXECUTION MODEL 36
4.2.2 CLASS TO THREADS MAPPING 36
4.3 ALGORITHMS 37

5 EARLY SCHEDULING ANALYSIS 41
5.1 ENVIRONMENT 41
5.2 APPLICATION 41
5.2.1 CLASS MAPPINGS 42
5.2.2 CLASS-TO-TREADS MAPPINGS 42
5.3 METRICS 43
5.4 WORKLOADS 44
5.5 RESULTS 45

6 EARLY SCHEDULING ENHANCEMENTS 51
6.1 BUSY-WAIT SYNCHRONIZATION 51
6.1.1 SAFETY AND LIVENESS 53

6.2 WORK-STEALING 53
6.2.1 STEALING WHEN QUEUES ARE EMPTY 54
6.2.2 STEALING WHILE SYNCHRONIZING 58

7 ENHANCEMENTS EVALUATION 67
7.1 EXPERIMENTS CONFIGURATIONS 67
7.2 SINGLE-SHARD 67
7.2.1 BUSY-WAIT RESULTS 68
7.2.2 WORK-STEALING RESULTS 69
7.2.3 BUSY-WAIT VS. WORK-STEALING 71
7.3 MULTI-SHARD 72

8 CONCLUSIONS 75
8.1 FUTURE WORK 75

REFERENCES 77

1. INTRODUCTION

The state machine replication (SMR) architecture is a common approach to provide fault tolerance to distributed asynchronous systems [31, 38]. The main idea is: server replicas execute client requests deterministically, in the same order. Consequently, replicas transition through the same sequence of states and produce the same sequence of outputs. An SMR can tolerate a configurable number of faulty replicas. Thus, application designers can focus on the inherent complexity of the application, while abstracting the difficulty of handling replica failures [20]. This approach has been successfully used in many contexts (e.g., [12, 21, 16]).

The natural SMR system model suggests a sequential architecture of server processes [38]. This approach, however, leads to poor system performance and low throughput, since increasing the number of replicas do not increase scalability, it only increases fault tolerance capacity. Out of this limitation, together with the proliferation of multi-core architectures, academia has turned to new approaches of SMR. Such approaches put aside the natural sequential paradigm and move on to applying parallelism concepts, either in the execution or control level.

A large number of techniques have been proposed in the literature to allow multi-threaded execution of requests at replicas (e.g., [22, 28, 29, 34]). Techniques that introduce concurrency in SMR are built on the observation that *independent* requests can execute concurrently while *conflicting* requests must be serialized and executed in the same order by the replicas.

Two requests conflict if they access common state and at least one of them updates the state, otherwise the requests are independent. Executing conflicting requests concurrently may result in inconsistent states across replicas. Hence, an important aspect in the design of parallel state machine replication (P-SMR) is how to schedule requests for execution on worker threads.

From the discussion above, the main correctness requirements is that conflicting requests must be serialized and executed in the same order across replicas. Regarding this aspect, the authors in [4] introduce a classification of scheduling approaches to Parallel SMR. Among the proposed classes, we studied solutions which fall in two of such classes:

- *Late Scheduling*: requests are scheduled for execution after they are ordered across replicas. Besides the aforementioned requirement on conflicting requests, there are no further restrictions on scheduling.

- *Early Scheduling*: part of the scheduling decisions are made before requests are ordered. After requests are ordered, their scheduling must respect these restrictions.

Since late scheduling has fewer restrictions, it allows more concurrency than early scheduling. However, the cost of tracking dependencies among requests in late scheduling may outweigh its gains in concurrency.

In CBASE [30], for example, each replica has a directed dependency graph that stores not-yet-executed requests and the order in which conflicting requests must be executed. A scheduler at the replica delivers requests in order and includes them in the dependency graph. Worker threads remove requests from the graph and execute them respecting their dependencies. Therefore, scheduler and worker threads contend for access to the shared graph. Not only is the graph a potential point of contention (i.e., scheduler and workers share the graph), but also graph updates introduce overhead. This is particularly problematic if the cost of request execution is low.

By restricting concurrency, scheduling can be done more efficiently. Consider a service based on the typical readers-and-writers concurrency model. A simple approach is to allow reads (i.e., requests that do not modify the state) to be scheduled on any worker thread, and write requests to be scheduled on all worker threads. To execute a write, all worker threads must coordinate (e.g., using barriers) to ensure that no request is ongoing and only one worker executes the write request.

This scheme is more restrictive than the one using the dependency graph because it does not allow concurrent writes, even if they are independent. However, previous research has shown that early scheduling can outperform late scheduling by a large margin, especially in workloads dominated by read requests [33, 34].

1.1 CONTRIBUTIONS

Although early scheduling shows performance gains, it can be improved to overcome its restrictions. In this work we describe the study, evaluation and enhancement of early scheduling in P-SMR, presenting the following contributions:

a. We have conducted a set of experiments with the early scheduling technique, with different configurations and workloads. In special, we evaluated the impact on thread behavior due to the imposed scheduling restrictions, analyzing levels of thread idleness and work distribution.

b. We propose enhancements to the basic early scheduling technique to overcome its restrictions, based on the observations from the experiments.

c. We have fully implemented the proposed enhancements and conducted a set of experiments to evaluate its performance.

d. We compared our techniques to the former early scheduling approach, to a sequential approach and to a late scheduling algorithm.

1.2 ORGANIZATION

This dissertation continues as follows. Chapter 2 introduces the system model and consistency criteria. Chapter 3 surveys related work. Chapter 4 introduces early scheduling. Chapter 5 evaluates early scheduling and presents a detailed analysis of restrictions imposed. Chapter 6 presents and discusses possible enhancements. Chapter 7 reports our experimental evaluation, with the basic model and enhancements. Chapter 8 concludes the work.

2. BACKGROUND

2.1 SYSTEM MODEL

We assume a distributed system composed of interconnected processes which communicate by exchanging messages. There is an unbounded set of client processes and a bounded set of replica processes. The system is asynchronous: there is no bound on message delays and on relative process speeds. We assume the crash failure model and exclude arbitrary behavior (e.g., no Byzantine failures). A process is *correct* if it does not fail, or *faulty* otherwise. There are up to f faulty replicas, out of $2f + 1$ replicas.

Processes have access to an atomic broadcast communication abstraction, defined by primitives *broadcast*(m) and *deliver*(m), where m is a message. Atomic broadcast ensures the following properties [18, 23][1]:

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Uniform Agreement*: If a process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity*: For any message m, every process delivers m at most once, and only if m was previously broadcast by a process.
- *Uniform Total Order*: If both processes p and q deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.

2.2 CONSISTENCY

Our consistency criterion is *linearizability*. A linearizable execution satisfies the following requirements [26]:

a. It respects the real-time ordering of operations across all clients. There exists a real-time order among any two operations if one operation finishes at a client before the other operation starts at a client.

b. It respects the semantics of the operations as defined in their sequential execution.

[1] Atomic broadcast needs additional synchrony assumptions to be implemented [14, 20]. These assumptions are not explicitly used by the protocols proposed in this work.

2.3 REQUEST INDEPENDENCE

To keep strong consistency, instead of sequentially executing requests, it has been observed that it suffices for a replica to execute sequentially only requests that access the same variables and one of the requests modifies the shared variables (conflicting or dependent requests). The other requests (independent requests) can be executed concurrently without violating consistency [38].

The notion of request interdependency is application-specific. Recently, several replication models have exploited request dependencies to parallelize the execution on replicas. More formally, request conflict can be defined as follows. Let R be the set of requests available in a service (i.e., all the requests that a client can issue). A request can be any deterministic computation involving objects that are part of the application state.

We denote the sets of application objects that replicas read and write when executing a request r as r's *readset* and *writeset*, or $RS(r)$ and $WS(r)$, respectively.

Definition 1. The conflict relation $\#_R \subseteq R \times R$ among requests is defined as

$$(r_i, r_j) \in \#_R \text{ iff } \begin{pmatrix} RS(r_i) \cap WS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap RS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap WS(r_j) \neq \emptyset \end{pmatrix}$$

Requests r_i and r_j *conflict* if $(r_i, r_j) \in \#_R$. We refer to pairs of requests not in $\#_R$ as *non-conflicting* or *independent*. Consequently, if two requests are independent (i.e., they do not share any objects or only read shared objects), then the requests can be executed concurrently at replicas (e.g., by different worker threads at each replica).

3. RELATED WORK

3.1 SCHEDULING

The scheduling research field has been active for several decades, with a large number of contributions. According to [32], one classification of scheduling algorithms regards a priori knowledge of the complete instance (i.e., set of tasks to schedule). With *offline* algorithms scheduling decisions are made based on the knowledge of the entire input instance that is to be scheduled. On the other hand, with *online* algorithms scheduling decisions are taken while tasks arrive. Regarding information about processing times, three main cases can be observed:

- *Deterministic:* when processing times are known a priori;
- *Stochastic:* when processing times can be assumed from known distributions;
- *Unknown:* when nothing can be assumed about the processing time of a task. Several algorithms of interest fall in this category, where processing times become known only when processing has completed.

Another criterion of classification is whether the scheduling is real-time or not. The primary concern of real-time scheduling is to meet hard deadline application constraints, which must be clearly stated, while non-real-time scheduling has no such constraints. According to these general aspects, the scheduling problem that we are addressing can be classified as online, without processing times information and non-realtime. A significant number of scheduling problems in the literature fall in this class. Among these, a further important aspect considered in the literature is whether tasks depend on each other, which restricts scheduling to respect such dependencies.

The scheduling of tasks with dependencies assuming a directed acyclic graph (DAG) topology is frequently considered in the literature. We can find early discussions on how to process tasks with such dependencies in several application areas. The scheduling of dependent computational tasks has been considered in different settings such as networks of workstations and clusters. More recently, with the proliferation of multicore architectures, it is also relevant in the scheduling of a DAG of interdependent tasks on such architectures (e.g., [36, 37, 42]).

However, the perspective taken in this work, as well as in several approaches to P-SMR, differs. The aim in this class of contributions is not only to parallelize the execution of requests, given their independence is known, but it includes the detection of their dependencies. The rate of incoming requests may vary according to the SMR application,

possibly achieving tens to hundreds of thousands of requests per second as reported in our experiments. As throughput increases, the overhead to manage dependencies gains in importance and may become a bottleneck in the system.

3.2 STATE MACHINE REPLICATION

In [4], a classification of approaches to Parallel SMR is introduced with the following classes:

- *Pipelined SMR* is a technique whereby replicas implement staging to enhance throughput. Replicas are organized in a series of modules connected through shared totally ordered message queues. Although staging improves the system throughput, there is always only one thread sequentially executing requests.

- *Late Scheduling* proposes that requests delivered at replicas be evaluated for conflict and scheduled concurrently for execution whenever they are independent from the ones under execution or pending. In CBASE [30], a parallel SMR is proposed where replicas are augmented with a deterministic scheduler. Based on application semantics, the scheduler serializes the execution of conflicting requests according to the delivery order and dispatches non-conflicting requests to be processed in parallel by a pool of worker threads. Since the scheduling decision is taken at the replica, before execution, the scheme has been dubbed late scheduling.

- *Early Scheduling* emerges from the observation that the overhead needed to keep command dependency information in late scheduling can be significant. Early scheduling (e.g., [5, 6]) trades concurrency for expeditious decisions at replicas. With application semantics, requests are grouped in classes and subsets of threads are assigned to implement classes. Threads to classes assignment is performed a priori. Since classes can conflict, requests from conflicting classes are serialized by involving threads from those different classes that synchronize to execute conflicting requests implemented by a thread level execution model. With this scheme, when requests arrive, the scheduler simply schedules them according to the mode (sequential or concurrent) to the set of pre-assigned threads. The complexity of dependency detection and according scheduling is thus bounded. In Chapter 4 we dive into more details about this technique.

- *Static Scheduling* is a more strict idea of Early Scheduling. It completely eliminates scheduling decisions at replicas. In [33] the authors adopt this approach. Clients map requests to different multicast groups based on request information which is application specific. Non-conflicting requests can be sent to distinct groups, while conflicting ones

are sent to the same group(s). At the replica side, each worker thread is associated to a multicast group and processes requests as they arrive. When a request arrives through more than one group, associated threads synchronize to execute, imposing an order on all involved threads (multicast groups).

Although the scheduling classification above encompasses several existing proposals to P-SMR, there are approaches to concurrent request execution in SMR-like architectures that do not fall into any of the identified categories. We call here SMR-like those architectures that depart from the principle of request independency and introduce additional cooperation among replicas, beyond the basic assumption of request ordering.

Rex [22] and CRANE [17] add complexity to the execution phase by introducing consensus about replicas synchronization events to solve non-determinism due to concurrency. Rex uses an execute-agree-follow strategy. A primary replica logs dependencies among requests during execution, based on shared variables locked by each request. This creates a trace of dependencies which is proposed for agreement with other follower replicas. After agreement replicas replay the execution restricted to the trace of the first executing server. CRANE [17] solves non-determinism with the input determinism of Paxos and the execution determinism of deterministic multi-threading [35]. CRANE implements an additional underlying consensus on synchronization events such that replicas see the same sequence of calls to synchronization primitives.

Eve [29] and Storyboard [28] use optimistic approaches that may lead to additional overhead in case replicas do not agree on the result. In Eve, this is done with optimistic execution and comparing results (consensus). If replicas diverge, roll-back and conservative re-execution is performed. With Storyboard, replicas have (a priori) forecasts of sequences of locks needed by requests. When execution deviates from expected, replicas have to establish a deterministic execution.

3.3 WORK-STEALING

The concept of work-stealing emerges within the context of parallel applications, which in turn come from the historical migration to multi-core architectures. Since the traditional approach of mono-processor systems has reached a certain technological limit, given that the traditional techniques of increasing the number of transistors, or clock frequency, cannot raise the performance threshold of single-core processors much further.

As a result, academia and the computer industry have accepted that the performance enhancement of future processors will be achieved on a large scale by increasing the number of processors (or cores) rather than trying to increase the performance of a single processor [41].

As such, parallel programming grows and brings with it greater concerns and challenges for programmers. It is relevant now to keep in mind more concepts inherent in parallelism, such as: process competition, concurrent data access control, the dependency between processes. Flow control of simultaneous executions, creation of new processes (threads) and their life cycle, among others, are also to be considered.

With the increase of concurrent threads in the same program, their allocation must also be considered. A good work balance must be ensured between threads and this is intrinsically linked to how they receive work to be executed. The process responsible for performing such task is called scheduler.

In parallel MIMD-style architectures, a scheduling algorithm must ensure that a sufficient number of concurrent threads are active to keep processors busy. Also, it should be concerned about keeping this number of threads not too high so as not to exceed memory limits. At the same time, it should also consider keeping related threads on the same processor, if possible, to minimize communication efforts between them.

According to [11], two scheduling paradigms have been consolidated in dealing with the scheduling problem of multithreaded systems: work-sharing and work-stealing. In work-sharing, when a process generates new threads, the scheduling algorithm attempts to migrate these threads to other processors, seeking to distribute the work among underutilized processors. In work-stealing, on the other hand, underused processors take the initiative to steal work from busy ones. Thus, thread migration occurs less frequently with work-stealing, since if processors keep busy, no threads are migrated by the work-stealing algorithm, but threads will always be migrated by a work-sharing algorithm.

The first ideas related to the concept of work-stealing can be attributed to [13] in parallel executions of functional programs. Moreover, it can be related to the implementation of Multilisp [24], which is an extension of the Lisp language with additional operators and semantics to handle parallel executions.

However, the first scheduler implemented with a randomized work-stealing algorithm for dependency multi-threaded applications was proposed by Blumofe and Leiserson in [11]. In this work, the authors prove the existence of optimal worst-case limits in relation to the expected execution time, space required and total communication cost. They have shown that the work-stealing technique has lower communication costs than the work-sharing approach.

They proposed an algorithm where each processor has a kind of double row (or deck) of threads. This deck has two ends: upper and lower. Threads can be inserted at the lower end and can be removed from both ends. A processor treats its own deck as a stack, inserting and removing threads from the lower end. Other processes, when stealing threads from decks that do not belong to them, do so from the top end. The authors proposed that the processor will work on its own tasks as long as possible. But it will steal work from a randomly chosen processor when its deck is empty.

Many variants of this algorithm were presented in the literature. In [39], for example, the authors present a threshold-based queueing model of shared-memory multiprocessor scheduling. Queueing model results in discrete state space and continuous-time Markov process. The large and complex state space is decomposed by assuming processor states stochastically independent and identical. It is approximate for finite number of processors, so compared with simulation. The general form allows modeling of degradation in system performance due to task migration. Even when migration costs are large, and contention compounds these costs by degrading system performance, task migration may still be beneficial. Threshold policies prevent processor thrashing: instability when tasks passed back and forth and most of system time spent on migration.

In [19] the authors argued that work-sharing outperforms work-stealing at light to moderate system loads, while work-stealing outperforms work-sharing at high loads, if the costs of task transfer under the two strategies are comparable. However, they argued that costs are likely to be greater under work-stealing, making work-sharing preferable. This is because work-stealing policies must transfer tasks which have already started to execute, while work-sharing policies can transfer prior to beginning execution.

In the context of affinity scheduling, the authors in [1] presented a work-stealing algorithm that uses locality information, and thus outperforms the standard work-stealing algorithm on benchmarks. Each process maintains a queue of pointers to threads that have affinity for it, and attempts to steal these first. They also bounded the number of cache misses for the work-stealing algorithm, using a “potential function” argument.

More theoretical results are presented from different perspectives, such as space bounds, strict multi-threaded computations, and stability. Work-stealing has also been investigated in a variety of other contexts. Among them, applications to thread scheduling, such as list scheduling [40], Fork–Join parallel programming [3], false sharing [15] and parallel batched data structures [2]. In [10] the authors improved the space bounds of Blumofe and Leiserson’s algorithm for a global shared-memory multiprocessor system. In [8] it is shown that the work-stealing algorithm is stable even under a very unbalanced distribution of loads.

4. EARLY SCHEDULING

Several approaches to P-SMR resort to application semantics to parallelize independent requests. While this allows concurrency, it introduces scheduling overhead to decide which requests are independent and which thread should execute each request. The early scheduling approach [5, 6] proposes a way to classify requests in *request classes* and a fast scheduling algorithm based on classes.

4.1 REQUEST CLASSES

The notion of request classes was introduced in [5] to denote application knowledge. Consider a service with a set R of possible requests. Each class has a descriptor and conflict information, as defined next.

Definition 2. Let R be the set of requests available in a service (same as considered in request conflicts). Let $C = \{c_1, c_2, ..., c_{nc}\}$ be the set of class descriptors, where nc is the number of classes.

We define request classes as $\mathcal{R} = C \rightarrow \mathcal{P}(C) \times \mathcal{P}(R)$,[1] that is, any class in C may conflict with any subset of classes in C, and is associated to a subset of requests in R. A conflict among classes happens when any two requests from those classes conflict, according to the conflict definition $\#_R$ from Chapter 2. Moreover, we introduce the restriction that a non-empty non-overlapping subset of requests from R is associated to each class.

Example. Consider a service partitioned in 2 shards where requests can be classified as read-only and read-write, per shard and globally. Different shards can be read and written independently. Read operations in a shard do not conflict. Writes conflict with reads and writes. Global writes conflict with any global or local operation. Global reads do not conflict with reads, global or local.

We model this application with the following classes. Read class C_{R1} in partition 1 conflicts with the write class C_{W1} on the same partition and with the global write class C_{Wg}. The read class C_{R2} in partition 2 conflicts with the write class C_{W2} on the same partition and with the global write class C_{Wg}. The class C_{Wg} also conflicts with itself, with the write classes and with the overall reading class C_{Rg}. Writing classes C_{W1} and C_{W2} also conflict with themselves and with the overall reading class C_{Rg}. Class C_{Rg} also conflict with itself. This is denoted in Figure 4.1, where classes are nodes and conflicts are edges.

To understand why class C_{Rg} is conflicting with itself, please refer to [6] where it is shown that such configuration generates more concurrency than if this class did not conflict with itself.

[1] We denote the power set of set S as $\mathcal{P}(S)$.

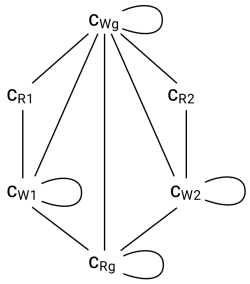


Figure 4.1 – Request classes definition with 2 shards.

4.2 CLASSES, THREADS AND EXECUTION MODEL

Central to the idea of early scheduling is that the scheduling algorithm avoids the late scheduling overhead, i.e., it does not have to evaluate every other pending request to decide how to schedule a new incoming one. It suffices to know the request's class to associate an appropriate worker thread. Thus, the scheduling overhead is bounded, independently of the population of pending requests.

4.2.1 EXECUTION MODEL

To accomplish such a straightforward scheduling algorithm, early scheduling adopts a replica execution model that will synchronize requests from conflicting classes. A replica will have one scheduler thread and *n* worker threads. Each worker thread has a separate input FIFO queue. The scheduler receives each request *r* totally ordered from consensus and decides to which worker thread(s) to associate.

a. If scheduled to one worker only, *r* can be processed concurrently with other requests.

b. If scheduled to more than one worker thread, then *r* depends on preceding requests assigned to these workers. Therefore, all workers involved in *r* must synchronize before one worker among these executes *r*.

4.2.2 CLASS TO THREADS MAPPING

With this execution model, the following class-to-thread-mapping rules must be applied to ensure linearizable executions:

i. *Every class is associated with at least one worker thread,* to ensure that requests are eventually executed.

ii. *If a class is self-conflicting, it is sequential.* Each request is scheduled to all threads of the class and processed as described in the previous section.

iii. *If two classes conflict, at least one of them must be sequential.* The previous requirement may help decide which one.

iv. *For conflicting classes c_1, sequential, and c_2, concurrent, the set of workers associated to c_2 must be included in the set of workers associated to c_1.* This requirement ensures that requests in c_2 are serialized w.r.t. c_1's.

v. *For conflicting sequential classes c_1 and c_2, it suffices that c_1 and c_2 have at least one worker in common.* The common worker ensures that requests in the classes are serialized.

These rules result in several possible class-to-threads mappings. This mapping problem was modeled as an optimization problem that must satisfy the previous rules and is optimized for the following conditions [6]: minimize threads in sequential classes and maximize threads in concurrent classes, assign threads to concurrent classes in proportion to their relative weight (i.e., the number of requests expected for these classes), and minimize unnecessary synchronization among sequential classes. A mapping is defined as follows.

Definition 3. $CtoT = C \rightarrow \{Seq, Conc\} \times \mathcal{P}(T)$ where: C is the set of class names; $\{Seq, Conc\}$ is the sequential or concurrent synchronization mode of a class; and $\mathcal{P}(T)$ the possible subsets of $T = \{t_0, .., t_{n-1}\}$, n is the number of worker threads at a replica.

Example. Following our example from Figure 4.1, considering 4 worker threads available, a possible mapping following the rules above is depicted in Table 4.1.

Table 4.1 – A possible mapping of 4 threads in Figure 4.1

$C =$	$\{seq, conc\}$	$\times \mathcal{P}$ (	$\{t_0,$	$t_1,$	$t_2,$	$t_3\}$	)
$C_{R1}=$	*conc*		$\{t_0,$		$t_2,$	$\}$	
$C_{R2}=$	*conc*		$\{$	$t_1,$		$t_3\}$	
$C_{W1}=$	*seq*		$\{t_0,$		$t_2,$	$\}$	
$C_{W2}=$	*seq*		$\{$	$t_1,$		$t_3\}$	
$C_{Rg}=$	*seq*		$\{t_0,$			$t_3\}$	
$C_{Wg}=$	*seq*		$\{t_0,$	$t_1,$	$t_2,$	$t_3\}$	

4.3 ALGORITHMS

With a *CtoT*, Algorithms 2 and 3 present the execution model for the scheduler and worker threads, respectively. Algorithm 1 presents some general definitions used by both algorithms, as well as by some of the algorithms described in Chapter 6. Whenever

a request is delivered by the atomic broadcast protocol, the scheduler (Algorithm 2) assigns it to one or more worker threads. If a class is sequential, then all threads associated with the class receive the request to synchronize the execution (lines 3–4). Otherwise, requests are associated to a unique thread (line 6), following a round-robin policy (function *PickInRoundRobin*).

Algorithm 1 General Early Scheduling Definitions used in Algorithms.

```
1:  constants:                                                  // knowledge provided for the scheduler
2:      R = {...}                                               // the set of all possible requests
3:      C = {c_1, ..., c_nc}                                    // the set of classes, constant
4:      T = {t_0, ..., t_nt-1}                                  // the set of threads, constant
5:      RC = C → (P(C) × P(R))          // for each class, the conflicting classes and requests it groups
6:      access functions:
7:          RC(c).conflicts : subset of classes conflicting with c
8:          RC(c).requests : subset of requests in c
9:      CtoT = C → ({Seq, Conc} × P(T))                         // the mode and the threads of a class
10:     access functions:
11:         CtoT(c).threads : subset of threads                 // associated with class c in CtoT
12:         CtoT(c).nThreads : cardinality of subset of threads // number of threads implementing class c
13:         CtoT(c).mode : Seq or Conc                          // mode of class c
14:         PickInRoundRobin(CtoT(req.class).threads)           // chooses next thread in round-robin
15: type:
16:     Request = [r, c] such that: r ∈ R ∧ c ∈ C ∧ r ∈ RC(c).requests
17:     access functions:
18:         req.class : for req ∈ Request, its class
19: shared variables:
20:     ∀t ∈ T
21:         t.queue ← ∅                                         // one input queue per worker thread
22:         access functions:
23:             t.queue.fifoPut() and t.queue.fifoGet(): FIFO produce in and consume from t.queue respectively
24:     barrier[C]                                              // one barrier per request class
25:     access function:
26:         ∀c ∈ C, barrier[c].arrive(): thread signals arrival to class' c barrier
```

Mutual exclusion while accessing above shared variables is assumed.

Algorithm 2 Early Scheduler.

```
1:  on deliver(Request: req):
2:      if CtoT(req.class).mode = Seq then                      // if execution is sequential
3:          ∀t ∈ CtoT(req.class).threads                        // for each conflicting thread
4:              t.queue.fifoPut(req)                            // synchronize to execute request
5:      else                          // else assign request to one thread of the class, in round-robin
6:          PickInRoundRobin(CtoT(req.class).threads).queue.fifoPut(req)
```

Each worker thread, as presented in Algorithm 3, takes one request at a time from its queue in FIFO order (line 4) and then proceeds depending on the synchronization mode of the class.

If the class is sequential, then the thread synchronizes with the other threads in the class using barriers before the request is executed (lines 6–12). In the case of a sequential class, only one thread executes the request. If the class is concurrent, then the thread simply executes the request (line 14).

Algorithm 3 Worker Threads for Early Scheduling.

```
 1: constant:
 2:     myId ∈ {0, ..., n − 1}                          // thisThread's id, out of n threads
 3: while true do
 4:     req ← thisThread.queue.fifoGet()                // wait until a request is available
 5:     if CtoT(req.class).mode = Seq then              // sequential execution:
 6:         if myId = min(CtoT(req.class).threads) then //   smallest id:
 7:             barrier[req.class].await()              //     wait for signal
 8:             exec(req)                               //     execute request
 9:             barrier[req.class].await()              //     resume workers
10:         else
11:             barrier[req.class].await()              //   signal worker
12:             barrier[req.class].await()              //   wait execution
13:     else                                            // concurrent execution:
14:         exec(req)                                   //   execute the request
```

Safety and *liveness* are argued in [6], where it is shown that these algorithms generate linearizable executions and that every request is eventually executed.

5. EARLY SCHEDULING ANALYSIS

Early scheduling restricts concurrency to allow fast scheduling decisions. This chapter evaluates early scheduling by analyzing the results from a set of experiments to understand how these restrictions affect thread utilization and load balancing. A more detailed presentation of these experiments of early scheduling in parallel state machine replication can be found at [7].

5.1 ENVIRONMENT

The experiments were conducted using seven computational nodes connected by a local-area network (cluster). Three server nodes implement BFT-SMaRt replicas, one per node. BFT-SMaRt [9] is a well-established library to develop SMR. It can be configured to use optimized protocols to tolerate both crash and byzantine failures. In our experiments, however, we consider only crash failures. BFT-SMaRt was implemented in JAVA programming language and uses an atomic broadcast protocol that executes a series of consensus instances to ordering sets of requests.

Each server node has the following configuration: AMD Opteron® Processor 6366 HE @ 2271.490Mhz, 32 physical cores (64 logical cores through the use of hyper-threading); 126GB RAM; Linux Ubuntu 4.15.0 operating system, 64 bits; Java Virtual Machine and OpenJDK version 11.0.3; OpenJDK 64-Bit Server VM.

Four client nodes were configured to run client processes. Each client node has the following configuration: Intel® Xeon® L5420 @ 2.50GHz processor with 8 physical cores; 8GB RAM; Linux Ubuntu 4.15.0 operating system, 64 bits; Java Virtual Machine and OpenJDK version 11.0.3; OpenJDK 64-Bit Server VM.

5.2 APPLICATION

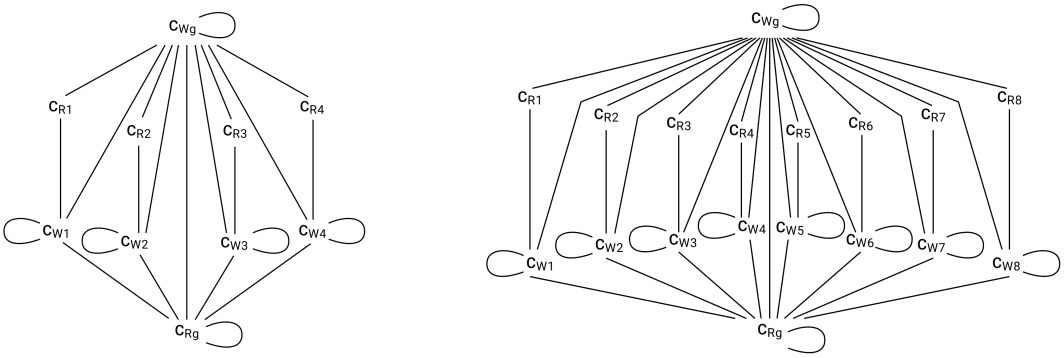
The experiments were performed using a linked list application. The application was implemented to support separate data shards, that is, each replica has an internal partitioned state. There are requests to read from the list and to write in the list, accessing either a single shard or all shards. A read operation checks whether an element is in one shard or in all shards, and a write operation includes an element in one shard or in all shards. Duplicated elements are not included in some shard, i.e., the write operation checks if some element already is in some shard before inclusion.

We conducted experiments with 2, 4 and 8 shards, in a system with 6, 10 and 18 request classes, respectively. In a deployment with n shards, there are n local (i.e., single-shard) reads classes, n local writes classes, one global (i.e., all shards) read class, and one global write class. Each replica was configured to run t worker threads, where each shard is assigned two threads (the read and write classes of each shard are mapped to the same two threads), and consequently, $t = 2n$.

5.2.1 CLASS MAPPINGS

Three types of class mappings were analyzed. The first is as presented in the example of Chapter 4, with 6 request classes.

The other two have 10 and 18 classes, respectively, and increased number of partitions, as shown in Figures 5.1(a) and 5.1(b). They respect the same conflict structure between read and write classes, as in the first mapping example of Figure 4.1. Local reads class C_{Rx} conflicts with global writes class C_{Wg} and with local writes class C_{Wx}, where x is the partition number. Local writes class C_{Wx} conflicts with itself, local reads class C_{Rx}, global writes class C_{Wg} and global reads class C_{Rg}. Global reads class C_{Rg} conflicts with global writes class C_{Wg} and with any local writes class C_{Wx}. Global writes class C_{Wg} conflicts with itself and any other class.



(a) 4 Shards (b) 8 Shards

Figure 5.1 – Request classes definition with 4 and 8 shards

5.2.2 CLASS-TO-TREADS MAPPINGS

Each replica was configured to run t worker threads, according to the class-to-threads mapping. In Tables 4.1 (from the example of Chapter 4), 5.1 and 5.2 we present

Table 5.1 – Threads to classes mappings for 4 shards and 8 threads

$C =$	$\{seq, conc\}$	$\times \mathcal{P}$ (	$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7\}$	)
$C_{R1} =$	*conc*		{		$t_2,$		t_4			}	
$C_{R2} =$	*conc*		$\{t_0,$						t_6	}	
$C_{R3} =$	*conc*		{			$t_3,$		t_5		}	
$C_{R4} =$	*conc*		{	$t_1,$						$t_7\}$	
$C_{W1} =$	*seq*		{		$t_2,$		t_4			}	
$C_{W2} =$	*seq*		$\{t_0,$						t_6	}	
$C_{W3} =$	*seq*		{			$t_3,$		t_5		}	
$C_{W4} =$	*seq*		{	$t_1,$						$t_7\}$	
$C_{Rg} =$	*seq*		$\{t_0,$				$t_4,$		$t_6,$	$t_7\}$	
$C_{Wg} =$	*seq*		$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7\}$	

Table 5.2 – Threads to classes mappings for 8 shards and 16 threads

$C =$	$\{seq, conc\} \times \mathcal{P}$	$(\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7,$	$t_8,$	$t_9,$	$t_{10},$	$t_{11},$	$t_{12},$	$t_{13},$	$t_{14},$	$t_{15}\})$
$C_{R1} =$	*conc*	$\{t_0,$	t_1														}
$C_{R2} =$	*conc*	{		$t_2,$	t_3												}
$C_{R3} =$	*conc*	{				$t_4,$	t_5										}
$C_{R4} =$	*conc*	{						$t_6,$	t_7								}
$C_{R5} =$	*conc*	{								$t_8,$	t_9						}
$C_{R6} =$	*conc*	{										$t_{10},$	t_{11}				}
$C_{R7} =$	*conc*	{												$t_{12},$	t_{13}		}
$C_{R8} =$	*conc*	{														$t_{14},$	$t_{15}\}$
$C_{W1} =$	*seq*	$\{t_0,$	t_1														}
$C_{W2} =$	*seq*	{		$t_2,$	t_3												}
$C_{W3} =$	*seq*	{				$t_4,$	t_5										}
$C_{W4} =$	*seq*	{						$t_6,$	t_7								}
$C_{W5} =$	*seq*	{								$t_8,$	t_9						}
$C_{W6} =$	*seq*	{										$t_{10},$	t_{11}				}
$C_{W7} =$	*seq*	{												$t_{12},$	t_{13}		}
$C_{W8} =$	*seq*	{														$t_{14},$	$t_{15}\}$
$C_{Rg} =$	*seq*	$\{t_0,$		$t_2,$		$t_4,$		$t_6,$		$t_8,$		$t_{10},$		$t_{12},$		t_{14}	}
$C_{Wg} =$	*seq*	$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7,$	$t_8,$	$t_9,$	$t_{10},$	$t_{11},$	$t_{12},$	$t_{13},$	$t_{14},$	$t_{15}\}$

each class-to-threads mapping used in the experiments, with t equal to 4, 8 and 16 threads, respectively.

5.3 METRICS

In order to evaluate the early scheduling performance with respect to its restrictions, we considered three distinct metrics:

a. *Synchronization idleness.* This metric represents the average wait time for a thread to synchronize with all other threads in the same class before executing a synchronizing request. It is obtained as follows:

   i High precision system nano-time is collected right before the first barrier *await* instruction, at lines 7 and 11 in Algorithm 3.

   ii In the thread responsible for executing the request, a second system time is collected right after the first barrier *await* instruction, before the *exec* instruction of line 8 in Algorithm 3.

   iii In waiting threads, the second measure of system time is collected after the second barrier *await* instruction of line 12 in Algorithm 3.

   iv The amount of waiting time (difference between the two instants of time collected as described above) is stored per second for each thread.

b. *Queue idleness.* This metric represents the average time that a thread waits for new requests in its queue. It is obtained as follows:

   i High precision system nano-time is collected right before the *fifoGet* instruction, at line 4 in Algorithm 3, which blocks the thread until a new request is available.

   ii The second system time is taken right after line 4 in Algorithm 3.

   iii The waiting time (difference between the two instants of time collected as described above) is stored per second for each thread.

c. *Queue size.* This metric represents the average size of a thread's queue. It is obtained by counting how many requests were returned by the *fifoGet* instruction, line 4 in Algorithm 3. In the implementation, this instruction actually returns a batch of requests available at the scheduler, in the same order as scheduled. The amount of requests is stored per second for each thread.

5.4 WORKLOADS

On the client side, we configured each node to run 10, 40 or 50 processes, according to the number of shards (2, 4 and 8, respectively), sending requests to servers with a mixed workload of read and write operations. Each client process sends batches of 50 operations per request, without any interval between each request. This configuration results in performance near its peak.

Several executions were performed submitting the application to different workloads, ranging the percentages of reads, writes, local and global operations. The percentage of writes and global operations ranged from low to high levels due to observation that as the level of conflicts increase, it directly affects the metrics that we are monitoring, especially thread synchronization idleness.

We also considered balanced and unbalanced workloads. In a balanced workload, each shard receives a similar number of local requests. In the skewed workload, each client process sends about 50% of its requests to only one shard (except for the experiments with 2 shards where one shard received 80%). The remaining requests are equally distributed across the remaining shards. We carried out experiments with different request execution costs, ranging from light, moderate to heavy costs (i.e., lists with 1K, 10K and 100K elements, respectively).

In these experiments, we represent different workloads with notation α-β-γ, where α is the percentage of local writes (i.e., writes in a single shard); β is the percentage of global operations (i.e., operations involving all shards); and γ is the percentage of global writes in global operations. For example, workload *25-5-25* has 25% of local writes and 5% of global

operations, where 25% of the global operations are global writes. Each experiment lasts 4 minutes, where results for the first minute are discarded (system warm-up). In the remaining three minutes, we collect data to compute average and standard deviation values for the metrics.

5.5 RESULTS

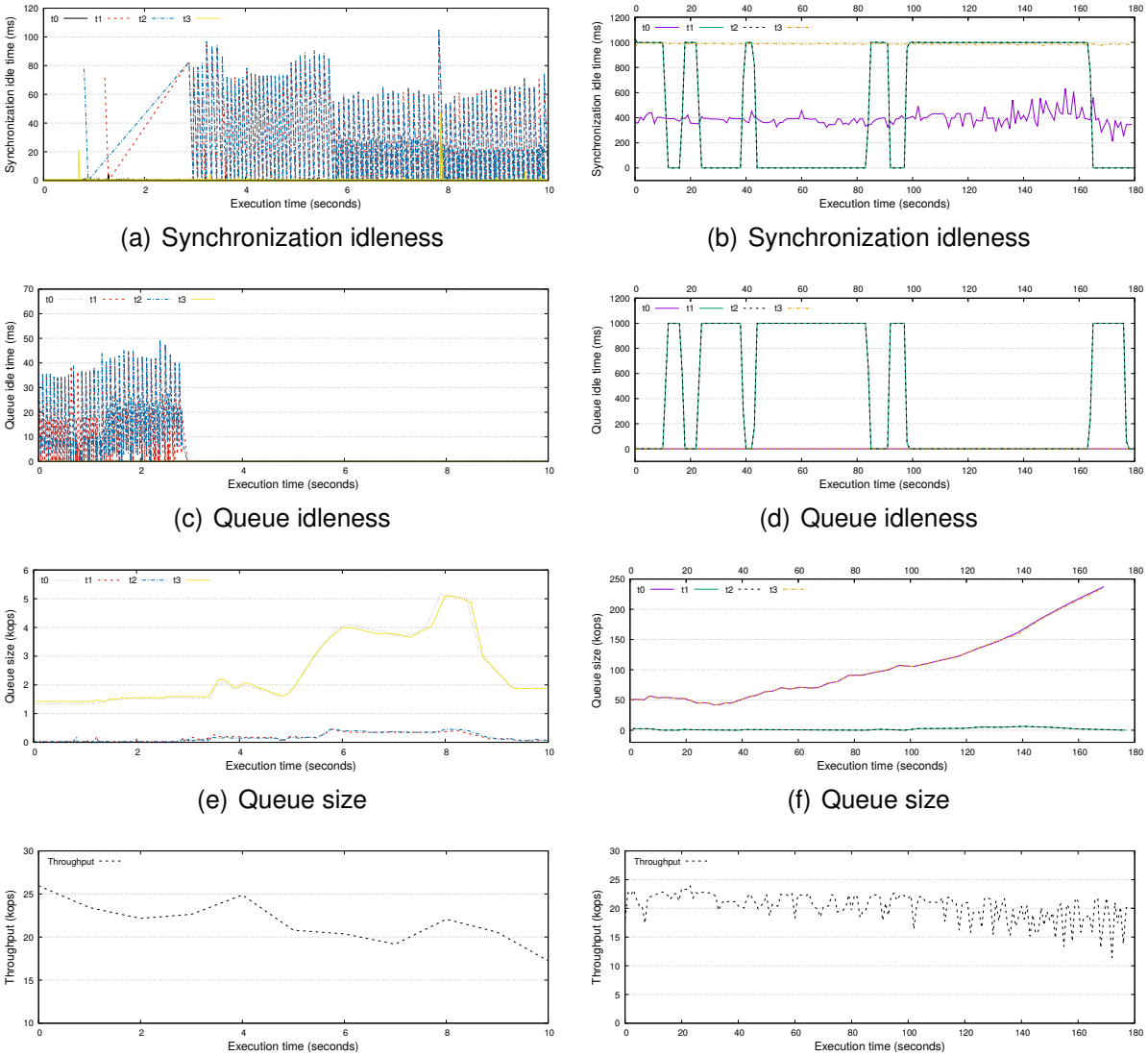
We start by presenting the results during a single execution with 2 shards, balanced workload with low level of conflicting requests and light operation costs to observe how the metrics evolve during execution. Due to the high amount of data collected in a run, and to understand how we thereafter consolidate the data, we first present only 10 seconds of a single execution. In Figure 5.2 (left), we can see thread behavior with respect to each metric and the system throughput, during this first short period of execution.

In this specific interval, we can observe that in the first 3 seconds there is a low rate of synchronization idleness. That happens due to high incidence of concurrent requests. This is also the reason why there is queue idleness, and low amount of requests in the queues. Since there are few requests in the queues, threads are more prone to wait for new ones. It incurs that system throughput is higher than in the next 7 seconds, when more sequential requests arrive, causing the threads to spend more time in the synchronization barriers. This increases the number of requests in the queues, decreasing the queue idleness (i.e., threads do not need to wait due to request availability in the queues), and decreasing the system throughput.

Based on these data, each metric was aggregated per second of execution. We present the results of an entire experiment execution in Figure 5.2 (right).

Synchronization idleness (Figure 5.2(b)): Each thread spends different amounts of time waiting for synchronization in the barriers. In this case of a balanced workload with light operation costs, threads $t0$ and $t3$ are the idlest. This behavior reflects the class-to-threads mapping (Table 4.1), where both threads are associated with a larger number of classes. Notice that thread $t3$ is the most idle because it never executes synchronized requests. This happens because $t3$ has the highest *id* (line 6 of Algorithm 3).

Queue idleness (Figure 5.2(d)): Queue idleness is inversely proportional to synchronization idleness. Threads $t1$ and $t2$ now are the most idle, due to faster execution of requests. This happens because both threads receive fewer requests, and more often need to get more requests from the scheduler. Thus, they are more prone to find their queues empty, resulting in waiting. Threads $t0$ and $t3$, however, do not need to wait since they spend much time in the barriers, waiting for sequential executions. This causes their queues to always have new requests to execute.



(a) Synchronization idleness (b) Synchronization idleness

(c) Queue idleness (d) Queue idleness

(e) Queue size (f) Queue size

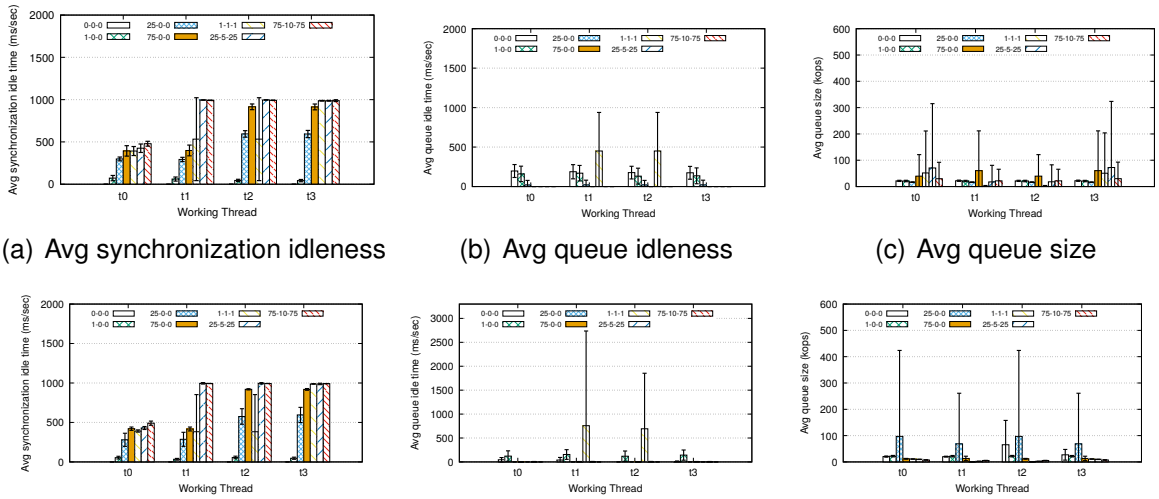
(g) Throughput (h) Throughput

Figure 5.2 – Metrics and throughput in 10 seconds (left) and an entire execution (right), balanced workload, low conflicts, 2 shards (4 threads), and light operations.

Queue size (Figure 5.2(f)): We can see how thread idleness impacts their accumulated work. As threads *t*0 and *t*3 are more often idly waiting for synchronization, the size of their queues keeps increasing during the execution. On the other hand, queue sizes of threads *t*1 and *t*2 are lower and more constant.

Figure 5.3 presents the consolidated results for a system with 2 shards, considering different workloads composed of light operations. We vary both the percentage of conflicts and request distribution among the shards.

Synchronization idleness (Figure 5.3(a)): We can observe in this experiment that idleness increases together with conflict percentage. Moreover, according to the classes to threads mappings (Table 4.1), thread *t*0 continued to be less idle than others (lowest *id* is always responsible for executing requests). Thread *t*3 is the most idle in the majority



(a) Avg synchronization idleness (b) Avg queue idleness (c) Avg queue size

(d) Avg synchronization idleness (e) Avg queue idleness (f) Avg queue size

Figure 5.3 – Results for 2 shards, 4 threads, light costs, with balanced workloads (top) and skewed workloads (bottom).

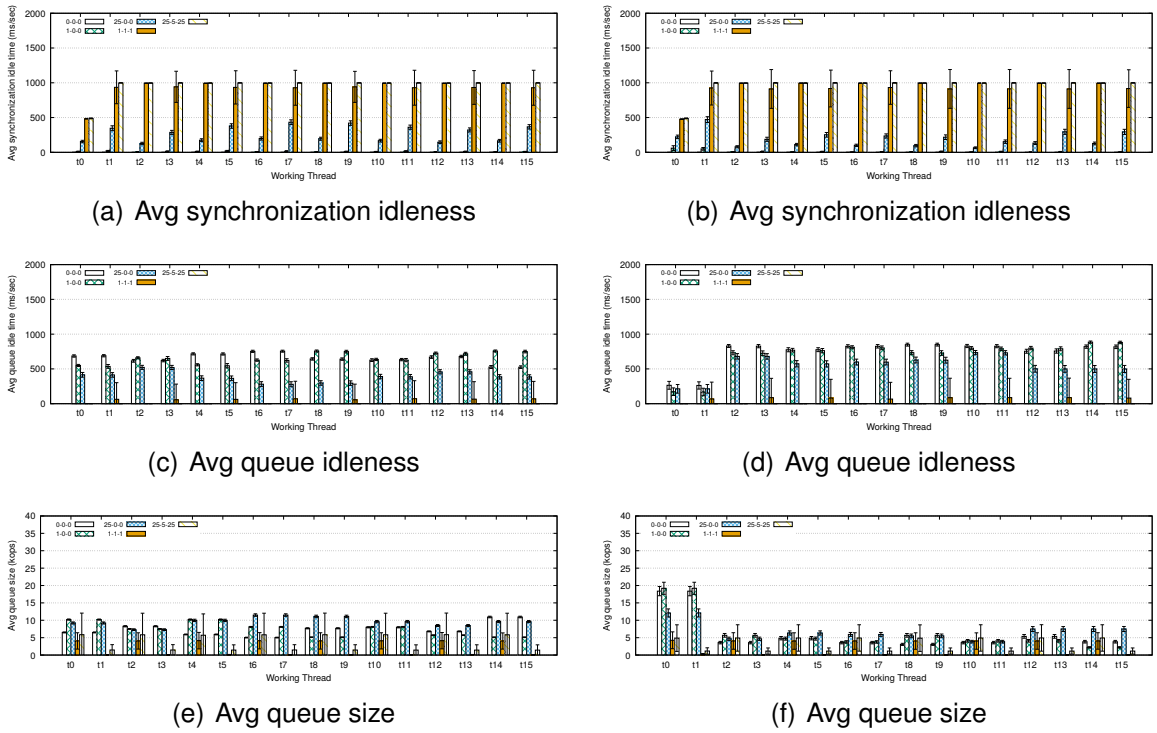
of workloads. However, for workloads with a high degree of conflicts (*25-5-25* and *75-10-75*), *t*0 executes most of the requests while the others remain almost all time only waiting for synchronizations. Notice that for workload *0-0-0* (only reads, which are concurrent requests) there is no synchronization idleness.

Queue idleness (Figure 5.3(b)): In general, the amount of queue idleness decrease with more conflicting workloads due to increasing synchronization idleness. While the percentage of conflicts in a workload gets higher, all threads spend more time in the synchronization barriers and. Consequently, more time is available for them to receive new requests in their queues, decreasing the time needed to wait for new requests.

Queue size (Figure 5.3(c)): This experiment shows that the difference between queue sizes among threads in the same workload increases in some cases, especially in cases with intermediary levels of conflict in the workload. This happens again because of the static classes to threads mappings. Notice the particular case of threads *t*1 and *t*2, which are associated with less amount of request classes and have, on average, fewer requests in their queues than the other two threads.

Skewed workloads: Figures 5.3(d), 5.3(e) and 5.3(f) present the results for same conflict percentages and shards/threads configurations but for skewed workloads. In this case, most of the requests are addressed to shard 1. This experiment shows that average thread idleness continues in high levels for most cases. It is, however, important to note the increasing in queue sizes variation and differences for most cases, where threads *t*0 and *t*2 accumulate more requests in their queues since they belong to the overloaded partition.

We can also observe high levels of standard deviation in some of the analyzed workloads. This happens because, depending on the demand from clients, in some measurement intervals the threads execute more sequential than concurrent requests, and vice-



(a) Avg synchronization idleness

(b) Avg synchronization idleness

(c) Avg queue idleness

(d) Avg queue idleness

(e) Avg queue size

(f) Avg queue size

Figure 5.4 – Results for 8 shards, 16 threads, light costs, with balanced workloads (left) and skewed workloads (right).

versa. Moreover, the synchronizations demanded to execute a sequential request is not needed for concurrent ones. This unbalance between sequential and concurrent requests execution leads to high variance.

Impact of the number of shards in the system: Figure 5.4 present the results for a system configured with 8 shards, considering balanced and skewed workloads where shard 1 received more requests. For better presentation, we exclude workloads *75-0-0* and *75-10-75* since they are the ones with most conflicting requests and always presented the same behavior with high levels of idleness.

In general, synchronization idleness again presented high levels for most workloads, and queue idleness increases for skewed workloads. It is also important to note that queue sizes suffer from more variation and differences in quantity among threads in the skewed workloads scenario. This behavior can be observed in Figure 5.4(f) where threads $t0$ and $t1$, associated to shard 1 (Table 5.2), receive more requests than all other threads. Consequently, their queues contain more requests to execute.

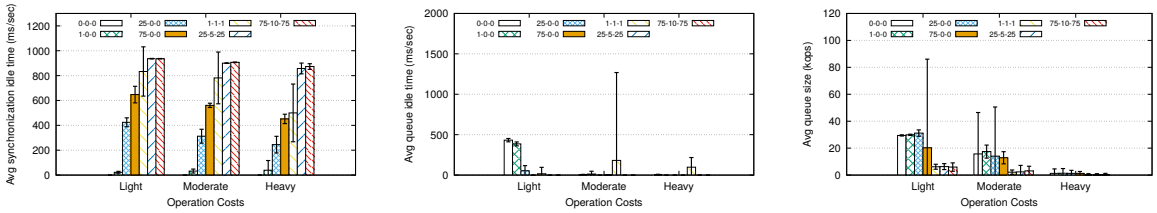
In this specific scenario with 8 shards and skewed workloads (Figure 5.4, right), we can observe how the static mappings of classes to threads affect performance. In these scenarios, thread $t1$ frequently has a larger amount of accumulated requests waiting for execution in its queue (Figure 5.4(f)). However, at the same time and for most of the workloads

considered, $t1$ together with almost all threads also present high idleness levels (Figure 5.4(b)).

Based on these results we can observe that a better distribution of work among the threads has the potential to improve system performance. For example, threads in idle states can receive requests originally designated, by the static mapping, to other overloaded threads. The main challenge is that this redistribution must respect all the conflict dependencies.

Impact of different execution costs (Figure 5.5): The final set of experiments studies how operation costs affect threads behavior, considering a system with 4 shards and 8 threads. We aggregated metrics averages of all threads, then we could range operation cost for all considered workloads.

Operation costs affect threads idleness since they spend more time executing heavier requests and, consequently, are less prone to become idle. Light operations incur in faster request execution, thus allowing threads to have more sequential requests to execute, increasing the amount of barrier synchronization.



(a) Avg synchronization idleness (b) Avg queue idleness (c) Avg queue size

Figure 5.5 – Results for 4 shards, 8 threads, different operation costs, with balanced workloads.

Figure 5.5(a) shows that thread synchronization idleness decreases slightly with higher operation costs. For queue idleness, we can observe the same phenomenon (Figure 5.5(b)). Finally, the queue size is smaller for higher operation costs (Figure 5.5(c)). This happens because servers need more time to process requests and send replies to clients. Consequently, clients remain most of the time blocked waiting for replies and few requests are issued in the system.

6. EARLY SCHEDULING ENHANCEMENTS

The Early Scheduling technique performs well for several workloads when compared to late scheduling. However, the experiments reported in Chapter 5 let us observe that threads may be idle (in some waiting point) while other threads have work to be done in their queues.

Out of these observations, we now present our proposed enhancements for the early scheduling technique. We investigated two approaches:

1. *Synchronization Mechanisms:* We propose an adaptation of the original early scheduling algorithm where the synchronization barriers (an expensive resource) were replaced by an alternative synchronization method. The new strategy keeps threads active, avoiding switching to system level, blocking, and thread re-scheduling. Assuming processors are available to the different worker threads, we evaluate the use of a busy-wait approach to thread synchronization.

2. *Work-Stealing:* We also propose a more complex implementation of the original early scheduling algorithm augmented with work-stealing techniques. In the context of early-scheduling, work-stealing is not trivial since the stealer threads have to enforce ordering in the place of the stolen ones, according to the classes and conflict definitions. Such restrictions introduce additional overhead to validate steal conditions and coordinate stealer threads.

6.1 BUSY-WAIT SYNCHRONIZATION

The busy-wait approach was based on the observation that barrier structures are a very expensive system resource that we want to avoid. We hypothesize that a large amount of the total thread synchronization idleness observed in experiments of Chapter 5 are related to system calls. As a result of increasing worker threads, there will be much more of such calls while threads are synchronizing using barriers. This phenomenon was already observed in [6].

Algorithm 4 General Busy-Wait Early Scheduling Definitions.

```
1: // extends definitions in Algorithm 1
2: shared variables:
3:     mark[c_1, ..., c_nc][2] ← [0, ..., 0][0, 0]      // one atomic integer request class, per cycle
4:     access functions:
5:         mark[c][cycle].get()                          // atomically read and return value
6:         mark[c][cycle].set()                          // atomically return and set the value, respectively
7:         mark[c][cycle].incGet()                       // atomically increment and return value
```

Therefore, we propose an adaptation of the original early scheduling worker thread execution model, as presented in Algorithm 5. This approach uses an array of atomic integers to synchronize threads, called *mark*, defined in Algorithm 4. The idea is that each thread involved in the execution of a request in class C must atomically increment a counter associated to C. The last thread to increment is responsible for executing the request and reset the counter. Other threads will wait until the counter resets.

Algorithm 5 Worker Threads For Busy-Wait Early Scheduling.

```
 1: constant:
 2:     myId ← id ∈ {0, ..., n − 1}                          // thisThread's id, out of n threads
 3: variables:
 4:     cycles[c_1, ..., c_nc] ← [0, ..., 0]                 // an array of cycles, one per request class
 5: while true do
 6:     req ← thisThread.queue.fifoGet()                     // wait until a request is available
 7:     if CtoT(req.class).mode = Seq then                   // sequential execution
 8:         cycles[req.class] = 1 − cycles[req.class]        // current synchronization cycle for req's class
 9:         if mark[req.class][cycles[req.class]].incGet() = CtoT(req.class).nThreads then   // signal arrival
10:             exec(req)                                    // if was the last to arrive, execute request
11:             mark[req.class][cycles[req.class]].set(0)    // signal execution is done
12:         else
13:             while mark[req.class][cycles[req.class]].get() ≠ 0 do   // wait in loop for request execution
14:
15:     else                                                 // concurrent execution:
16:         exec(req)                                        // execute the request
```

However, a simple array of atomic integers per class is not sufficient to guarantee progress. To understand why, suppose 2 threads, t_1 and t_2, are associated by the mapping to synchronize requests of class c_1. Consider a scenario where two requests of the same class c_1 are contiguously assigned to threads t_1 and t_2. Then, the following execution steps can take place: thread t_1 and t_2 pick the first request of class c_1 for execution. From this point, both verify it is a sequential request and increment the atomic counter associated to class c_1.

Suppose thread t_1 was the last to increment and thus becomes responsible for execution, but is preempted before execution is done. Then, as thread t_2 is not responsible for executing, it will wait in the loop, but suppose it is preempted before checking if the counter has value 0. Then, thread t_1 re-assumes and finishes execution of the first request, sets the counter to zero and continues to pick the second request of the same class c_1 from its queue. It then increments the atomic counter again, for the same class. Now it was the first thread to increment, and it does not become responsible for execution. At this point, both threads t_1 and t_2 are waiting for someone to reset the counter to zero (since t_2 missed the reset related to the first request), which will never happen.

Therefore we introduce cycles per class, as presented in Algorithm 5. Threads belonging to a class are either in cycle 0 or 1 for that class, and each time a request from that class is executed, threads switch cycle to execute a new request. When all threads belonging to the class have entered the same cycle, then the request can be executed. This

prevents the situation above discussed, since the new cycle has a new variable to control the number of threads joined. Only two cycles are needed because the request in the new cycle will only execute when no thread is in the former one.

No modification in the scheduler algorithm is needed for this approach. However, it is not common sense to assume an available processor for each worker thread. Moreover, it is not recommended to hold an expensive system resource consuming all power it can get by doing nothing. Therefore, a better utilization of such useless processing could be more profitable, as for example stealing work.

6.1.1 SAFETY AND LIVENESS

Safety: the barrier in the original Early Scheduling worker thread (Algorithm 3) ensures that all involved threads synchronize to execute the sequential request and do not advance before finishing its execution. The barrier is substituted here. The sequential request is executed when all involved threads reached the request in their input queues since the atomic variable is only incremented once by each thread and only the last thread to increment executes the request. After executing, the thread signals the other ones to stop waiting. Thus, the mechanism keeps the same barrier properties during request execution: one thread executes and all other ones (a) have arrived to the request in their queues and (b) wait for the request to finish before proceeding.

Liveness: for a given class, all its threads initiate in cycle 0 and deterministically switch to the next when a sequential request is processed. Since all threads have the same requests of their class in the input queue, eventually all will switch to the next cycle and complete the number of threads to execute the request. Moreover, since all threads have the same order of common requests, they will not build cycles while synchronizing to execute different requests. Thus, the synchronization mechanism does not block.

6.2 WORK-STEALING

The idea of adapting the early scheduling algorithm to support work-stealing techniques emerged from the results observed in Section 5.5 of Chapter 5. These results reported the co-existence of thread idleness and unbalanced work. We could relieve this phenomenon by enabling some threads to steal work from overloaded ones while waiting both for new requests or in the synchronization process.

To achieve such an adaptation, we must fulfill some conditions to ensure correctness while stealing. Work-stealing cannot violate the order of conflicting requests. However,

requests from a thread *v* (from now on called *victim*) could be stolen and executed concurrently by thread *s* (from now on called *stealer*) without compromising correctness if requests under execution by *v* and *s* are independent. Therefore, to keep safety we have to argue for order preservation. To complement, we have to argue that work-stealing mechanisms will not cause deadlock.

To describe the work-stealing strategies developed we define the following 5 main characteristics, which may be instantiated for each different algorithm:

- *Stealing condition:* defines conditions necessary to enable a stealer *s* to steal from a victim *v*, both in the state of the threads as well as the workload. These conditions alone are not sufficient and work with the other characteristics.
- *Stealing points:* defines the exact points of the original early scheduling algorithm where threads will proceed to stealing mechanisms;
- *Who is stolen:* defines which threads are victims of stealing, and on which order;
- *Number of occurrences of steal:* related to the above, defines how many times a stealer will try to steal work from other threads;
- *How much work is stolen:* defines how much work will be stolen.

6.2.1 STEALING WHEN QUEUES ARE EMPTY

Out of the observation from experiments of Chapter 5, threads can become idle while waiting for new requests in their queues. Hence, the first and most simple idea is to steal work from other threads while waiting for the arrival of new requests. In this approach, the stealing conditions and properties are instantiated as follows:

- *Stealing condition:* only concurrent requests can be stolen. The victim must not be executing a sequential request, neither have a sequential request in its queue;
- *Stealing points:* the stealer must be waiting for new requests in its own execution queue;
- *Who is stolen:* all threads, except the stealer itself, will be victims of stealing. They will be chosen by their *id*, starting from the smallest. If a thread can not be stolen, next *id* will be tried;
- *Number of occurrences of steal:* a stealer will repeatedly attempt to steal until new requests arrive at its own execution queue.

- *How much work is stolen:* all concurrent requests present in the victim's queue.

Algorithms 7 and 8 show the early scheduling based execution model of the work-stealing adaptation. Algorithm 6 shows some general definitions used by both work-stealing algorithms.

Algorithm 6 General Work-Stealing Definitions.

```
1: // extends definitions in Algorithm 1
2: shared variables:                                  // assumed consistent under concurrent manipulation
3:    ∀t ∈ T
4:       readyQueue ← ∅                               // a separate queue with requests ready for execution
5:       execQueue ← ∅                                // separate queue with requests under execution
6:       readyFlag ← 0                                // atomic flag: 1 if there is a seq req in readyQueue; 0 otherwise
7:       execFlag ← 0                                 // atomic flag: 1 if there is a seq req in execQueue; 0 otherwise
8:       marker[t_0, ..., t_{n-1}] ← [0, ..., 0]      // atomic array: 1 at entry s if thread s stole from t and not finished yet
```

Algorithm 7 Work-Stealing Scheduler.

```
1: on deliver(Request: req):
2:    if CtoT(req.class).mode = Seq then                          // if execution is sequential
3:       ∀t ∈ CtoT(req.class).threads                             // for each conflicting thread
4:          atomic:
5:             t.readyQueue.fifoPut(req)                          // synchronize to exec req
6:             t.readyFlag ← 1                                    // signal there is a sequential request in t's readyQueue
7:    else                                                        // else assign req to one thread in round-robin
8:       PickInRoundRobin(CtoT(req.class).threads).readyQueue.fifoPut(req)
```

The scheduler (Algorithm 7) inserts requests in a new separate queue of requests ready for execution, called *readyQueue* (lines 5 and 8). It also updates a flag in each thread (line 6) when assigning them with sequential requests. This flag holds the information whether there is a sequential request in the *readyQueue* of thread *t* (the need for a new queue will be explained next).

We can see in Algorithms 6 and 8 that each worker thread *t* is augmented with two separate queues: *readyQueue* and *execQueue*. The *readyQueue* holds requests assigned by the scheduler but not yet transferred to the *execQueue*. The *execQueue* holds requests under execution by thread *t*. Each thread has an array of atomic flags (called *marker*) indicating the stealing relations among threads. If a victim thread *v* has value 1 at entry *s*, it means that stealer thread *s* has stolen work from *v* and has not finished the execution yet, otherwise the value is 0. Hence, when thread *v* is about to execute sequential requests, it needs to verify these flags to find whether it was stolen and needs to wait for the stealer to finish (Algorithm 8, lines 15–16).

A stealing attempt will take place if a stealer *s* finds its queue empty. It calls method *Steal*() which atomically verifies the stealing conditions for each possible victim *v*, except itself. If the conditions are satisfied, it atomically steals all requests from *v*'s *readyQueue* and sets the marker of thread *v* indicating that it has been stolen. Then, it executes the stolen requests. Once finished, it signals victim *v* and stops the loop, trying its own queue again.

Algorithm 8 Work-Stealing Algorithm For Each Worker Thread t.

```
 1: constant:
 2:     thisThread ∈ T                                                    // the current thread
 3: Worker-Thread t is as follows:
 4:     while true do
 5:         atomic:
              // thisThread. is implicit if not stated otherwise.
 6:           execQueue ← readyQueue          // transfers all requests from readyQueue to execQueue
 7:           execFlag ← readyFlag          // update flag which may indicates execution of a sequential req
 8:           readyQueue ← ∅                                                // clear readyQueue
 9:           readyFlag ← 0                  // signal there is no sequential requests in readyQueue
10:         endAtomic
11:         if execQueue ≠ ∅ then                                          // if there is something to execute:
12:           while req ← execQueue.fifoGet() do
13:             if CtoT(req.class).mode = Seq then
14:               if thisThread = min(CtoT(req.class).threads) then
15:                 for all s ∈ T \ {thisThread} do          // verify if some stealer thread stole from me
16:                   wait until marker[s] = 0                // atomically reads marker. Wait until is 0
17:                 barrier[req.class].await()
18:                 exec(req)
19:                 barrier[req.class].await()
20:               else
21:                 barrier[req.class].await()
22:                 barrier[req.class].await()
23:             else
24:               exec(req)
25:         else                                                  // no requests available. Will try to steal
26:           Steal(thisThread)
27: procedure Steal(s ∈ T)                                                 // s is the stealer thread
28:     for all v ∈ (T \ {s}) do          for all victim threads, starting from id 0, except the stealer
29:         atomic:
30:           if v.readyFlag = 0 ∧                                  // only steals concurrent requests
31:             v.execFlag = 0 ∧                  // if victim not executing sequential requests
32:             v.readyQueue ≠ ∅ then                                 // there is something to steal
33:               s.execQueue ← v.readyQueue          // steal all requests from victim's readyQueue
34:               v.readyQueue ← ∅                                    // clear victim's readyQueue
35:               v.marker[s] ← 1                                      // signal s stolen from v
36:         endAtomic
37:         if s.execQueue ≠ ∅ then                                    // steal succeeded, execute
38:           for all req in s.execQueue do
39:             exec(req)                                           // execute all stolen requests
40:           v.marker[s] ← 0                          // atomically signal finished execution
41:           break for all                  // steals and executes once, then tries it's own again
```

6.2.1.1 *Safety Discussion - Order Preservation*

We have to argue that Algorithms 7 and 8 preserve the order of conflicting requests. Said differently, they can only commute independent requests.

The algorithms impose conditions on the contents of *readyQueue* and *execQueue* using *readyFlag* and *execFlag* that mark if the respective queues have conflicting requests. Notice that queues and flags are accessed in the same atomic blocks. This ensures that the flags are consistent with the respective queue's contents.

The steal procedure states that both *readyFlag* and *execFlag* of the victim have to be 0, that is, both queues have only concurrent requests, to allow the steal to take place. When this happens, the victim continues execution while the stealer starts processing the stolen requests. From this point, we have two possibilities: either the stealer thread or the victim finishes processing its *execQueue* first. The first case is simple: independent requests where finished concurrently by the stealer under the conditions imposed and the victim proceeds to process normally. In the second case the victim will process new incoming requests draining its *readyQueue* again. If the new incoming requests are again concurrent, then they can be processed concurrently with the stealer. Otherwise, to process a conflicting request the stealer has to finish first. This is ensured in line 16 of Algorithm 8, stating that the victim will wait for all stealers to finish before proceeding to the conflicting request. The stealer *s*, when finishing processing will signal on the specific victims *v* marker (*v*.*marker*[*s*]).

The above ensures that no conflicting requests are commuted, either by preventing steal to take place or by having the victim await stealers to finish before processing a conflicting request enqueued after the stolen ones - which are the only cases possible.

6.2.1.2 *Liveness Discussion - No Deadlock*

By construction, we can observe in Algorithm 8 that, once a thread *s* steals from *v*, *s* unconditionally processes the whole contents of its *execQueue*. Also, we observe that a victim, when processing its *execQueue*, either proceeds independently or it awaits stealer threads to finish. Since stealer threads unconditionally process their contents, eventually the victim will proceed to process its *execQueue*. Also, notice that if the victim is awaiting for a stealer s_1 to finish it is because its ready flag is set (there is a conflicting request) and thus it will not become a victim of another stealer s_2. This ensures that once a victim has a conflicting request to process, eventually all current stealers will have finished their stolen works and that no new steal attempt will succeed. Therefore the victim is ensured to make progress.

A stealing cycle among threads is possible but does not deadlock. Suppose t_1 has empty *readyQueue* and tries to steal from t_2 which is executing from its *execQueue* and has items in *readyQueue*. t_1 becomes stealer and t_2 victim. Now suppose that t_2 finishes its *execQueue*, finds its *readyQueue* empty, tries to steal from t_1. t_1 is processing stolen work from t_2 but in the meanwhile its *readyQueue* is populated. t_2 steals from t_1. We have a stealing cycle. Both threads nonetheless will make progress since by construction the stolen work is independent and therefore unconditionally processed. While non-conflicting commands are issued, threads can freely steal from each other as stealers are idle (depending on the workload, this process may cause threads just to switch work, in other cases better balancing could be achieved). When a conflicting command is issued to a thread, it cannot become a victim anymore and will process its queue on its own.

6.2.2 STEALING WHILE SYNCHRONIZING

Although the approach presented before introduces the concept of stealing work from possibly overloaded threads, it is restricted to the situation where there are no requests available for standard execution. The next step is to extend the algorithm to afford stealing also while waiting for synchronizing execution.

In order to increase the possibility of successful stealing attempts, and based on the results of Chapter 5 which have shown the high levels of thread idleness during synchronization, we need to provide a way for threads to steal also when idle for synchronization of sequential requests.

6.2.2.1 *Semi-blocked synchronization*

To provide stealing during synchronization process, it is necessary to replace the first barrier blocking step from the basic early scheduling approach (lines 17 and 21 of Algorithm 8), for a non-blocking synchronization mechanism.

We can use atomic variables to synchronize threads. Using an array of atomic integers per class, we can provide a way for threads to record an arrival at the point of execution of a sequential request for a specific class, without being blocked, by simply increment the atomic variable, as a counter. Such a strategy allows us to identify the last thread to arrive and orientate it to execution, while previous threads are orientated to stealing procedures, until the execution is done. After the executor thread finishes execution, it signals the other threads and proceeds to the barrier, waiting for synchronization before proceeding to standard execution. The stealers keep checking if the execution finished, and once it does, proceed to the barrier to wait for synchronization.

Using this strategy, we implemented a new version of the work-stealing algorithm. However, it was necessary to provide stronger conditions to ensure correctness and linearizability, as described next.

- *Stealing points:* the stealer must be waiting for new requests in its own execution queue; or waiting for synchronization of a sequential request - this second point is introduced in this algorithm;
- *Stealing condition:* depends on the stealing point. The same conditions of the previous work-stealing algorithm are kept if the stealer is waiting for new requests. For the second stealing point: while synchronizing for request of class C_1, stealer thread s can not steal requests from any class C_2, from any victim thread v, if C_1 and C_2 conflict. This prevents the stealer thread do commute order of conflicting requests.
- *Who is stolen:* same as previous work-stealing algorithm;

- *Number of occurrences of steal:* a stealer will repeatedly attempt to steal until either: 1) new requests arrive to its own execution queue, in case when it proceeded to steal because its queue was empty; or 2) the sequential request execution is done, in case it proceeded to steal when waiting for synchronization of a sequential request.

- *How much work is stolen:* same as previous work-stealing algorithm.

Algorithms 10 and 11 show the adapted execution model of semi-blocked work-stealing. We called it semi-blocked because it still has a blocking step. Algorithm 9 shows some general definitions used by the semi-blocked work-stealing algorithms.

Algorithm 9 General Semi-Blocked Work-Stealing Definitions.

```
1: // extends definitions in Algorithm 6
2: shared variables:                                   // consistent under concurrent manipulation
3:     ∀t ∈ T
4:         readyClasses[C] ← [0, ..., 0] // 1 at entry c means t.readyQueue contains req of class c. 0 otherwise
5:     syncMark[c_1, ..., c_nc] ← [0, ..., 0]                    // one atomic integer request class
6:     access functions:
7:         syncMark[c].get()                                     // atomically read and return value
8:         syncMark[c].set()                  // atomically return and set the value, respectively
9:         syncMark[c].incGet()                                  // atomically increment and returns value
```

Algorithm 10 Scheduler For Stealing while Synchronizing.

```
1: on deliver(Request: req):
2:     if CtoT(req.class).mode = Seq then
3:         ∀t ∈ CtoT(req.class).threads
4:             atomic:
5:                 t.readyQueue.fifoPut(req)
6:                 t.readyFlag ← 1
7:                 t.readyClasses[req.class] ← 1          // signal t's readyQueue now contains req's class
8:         endAtomic
9:     else
10:        PickInRoundRobin(CtoT(req.class).threads).readyQueue.fifoPut(req)
11:        t.readyClasses[req.class] ← 1                  // signal t's readyQueue now contains req's class
```

Algorithm 10 extends the functionality of Algorithm 7 by controlling a new structure in each thread called *readyClasses*. This structure is an array indexed by request classes to represent the classes of requests present in the *readyQueue*. After assigning any request r from class c to thread t, the scheduler updates the position in t.*readyClasses* indexed by the respective request class c. The value 1 at entry c in t.*readyClasses* means that t.*readyQueue* contains a request of class c. Value 0 means that no request of class c is present in t.*readyQueue*.

The Algorithm 11 is the worker thread that extends functionality of Algorithm 8. It accesses a shared array (called *syncMark*) of atomic counters, one for each class, as well as the same barriers array, one for each class, as the previous algorithm. Whenever a sequential request is about to be executed, all threads signal arrival at the execution point

Algorithm 11 Worker Thread t for Semi-Blocked Work-Stealing

```
1: constant:
2:     thisTh ∈ T                                                    // the current thread
3: Worker-Thread t is as follows:
4:     while true do
5:         atomic:
6:             execQueue ← readyQueue
7:             execFlag ← readyFlag
8:             readyQueue ← ∅
9:             readyFlag ← 0
10:            readyClasses ← [0, ..., 0]        // clear readyClasses when transfering requests to execQueue
11:        endAtomic
12:        if execQueue ≠ ∅ then
13:            while req ← execQueue.fifoGet() do
14:                if CtoT(req.class).mode = Seq then
15:                    for all s ∈ T \ {thisTh} do
16:                        wait until marker[s] = 0
17:                    if syncMark[req.class].incGet() = CtoT(req.class).nThreads then        // signal arrival
18:                        exec(req)                                  // last thread to arrive executes
19:                        syncMark[req.class].set(0)                 // signal execution is done
20:                        barrier[req.class].await()                 // synchronizes after execution
21:                    else
22:                        while syncMark[req.class].get() ≠ 0 do     // while execution not finished
23:                            Steal(thisTh, req.class)               // attempt to steal, informing current class
24:                        barrier[req.class].await()                 // synchronizes after execution
25:                else
26:                    exec(req)
27:        else
28:            Steal(thisTh, null)                                    // attempt to steal, informing no class
29: procedure Steal(s ∈ T, c ∈ C)   // s is the stealer thread, c is null or the class of seq req being executed
30:    for all v ∈ (T \ s) do
31:        atomic:
32:            if v.readyFlag = 0 ∧
33:               v.execFlag = 0 ∧
34:               ( c = null ∨                                        // no class was informed, or
35:                 NoConflict(v, c) ) ∧                              // no conflict with class c
36:               v.readyQueue ≠ ∅ then
37:                s.execQueue ← v.readyQueue
38:                v.readyQueue ← ∅
39:                v.marker[s] ← 1
40:                v.readyClasses ← ∅                  // if steal succeeds, clear victim's readyClasses
41:        endAtomic
42:        if s.execQueue ≠ ∅ then                                    // if steal succeeded
43:            for all req in s.execQueue do
44:                exec(req)
45:            v.marker[s] ← 0                                        // atomically signal finished execution
46:            break for all          // steals and executes once, then tries it's own queue; or verifies mark
47: procedure NoConflict(v ∈ T, c1 ∈ C)   // v is the victim thread, c1 is the class of seq req being executed
48:    for all c2 ∈ RC(c1).conflicts do                               // for each class c2 conflicting with c1:
49:        if v.readyClasses[c2] = 1 then                             // if it is present in v.readyQueue, then:
50:            return false                                           // return false
51:    return true                    // if no conflicting class is included in v.readyQueue, return true
```

of the respective request class. The last thread to arrive is responsible for executing the request. Other threads are free to attempt stealing while the executor has not finished yet.

The *Steal*() procedure implements the same stealing algorithm presented in Algorithm 8, yet augmented with a new restriction. If parameter *c*, which is the class of the request being synchronized for execution, is informed (value not null), the stealer *s* attempting to steal from victim *v* is not allowed to steal requests from any class that may conflict with *c*. This is ensured with the *NoConflict* procedure that checks the conflicts of the request classes. In the case of any conflicting class, steal does not succeed. If the steal succeeds, *s* must atomically clear *v*.*readyClasses*.

Safety Discussion - Order Preservation

We argue that Algorithms 10 and 11 preserve the order of conflicting requests. As in the first work-stealing execution model (Algorithms 7 and 8) they can only commute independent requests. The new algorithms impose the same conditions on the contents of *readyQueue* and *execQueue* using the same flags. Hence, they also ensure that the flags are consistent with the respective queue's contents.

The first stealing point is handled as the previous algorithm and will not be discussed. For the second stealing point, the work stolen cannot conflict with the stealer's ongoing synchronization class. As this strategy supports non-blocking arrival at the execution point, there are two possibilities: all threads, except the last to arrive, enter the stealing block (line 22), and the last thread enters the execution block (line 18). From this point, one thread executes the sequential request, and other threads perform steal attempts. When stealer *s* attempts to steal from victim *v*, and *s* is waiting for execution of a request of class c_1, the steal can not succeed if *v*.*readyQueue* contains class c_2 that conflicts with c_1. To ensure that, the stealer atomically verifies the values of *v*.*readyClasses* with respect to c_1, aborting the steal if it finds a conflict. This ensures that no conflicting requests are commuted by preventing the steal in face of a conflict.

Both execution flows stop at the blocking step (method *await*) at lines 20 and 24 of Algorithm 11, ensuring no conflicting request is commuted.

Liveness Discussion - No Deadlock

We argue by stealing points. The first point, when the stealer's *readyQueue* is empty, was argued in the previous algorithms. We have to discuss the second stealing point.

Recall that due to the stealing conditions, only victim's concurrent requests, when the victim is processing concurrent requests, can be stolen and processed by a stealer. From the victim's point of view, the order is not violated due to the nature of its requests. The stealing conditions prevent a stealer from stealing requests that conflict with the request it is currently awaiting for synchronization (second stealing point). In such case it cannot

steal because it cannot tell the right order among them, this has to be enforced by the early scheduling execution model.

Therefore, the same arguments as in the previous algorithm apply: a stealer executes unconditionally; eventually all stealers of a victim finish; if the victim has concurrent requests in its *readyQueue* it continues processing (and possibly being victimized by other stealers); if the victim has a sequential requests, stealers cease to steal, finish their current stolen works, and the victim synchronizes for the sequential request as stated in the early scheduling execution model.

6.2.2.2 *Barrier-free synchronization*

Although the previous approach improves stealing, it is still restricted with a last barrier blocking step (lines 20 and 24 of Algorithm 11). To further enhance this approach, we extended the algorithm to eliminate all barrier structures, reaching the same baseline of concurrency control as the busy-wait algorithm.

In order to eliminate all the barriers we must rely only on atomic variables to synchronize threads. Instead of having arrays of barriers per class, we could augment the current array of atomic counter of threads per class. We have already discussed in Section 6.1 that a simple array is not sufficient to ensure progress, and thus we need a bi-dimensional array (a matrix) to introduce cycles per class.

Algorithm 12 General Barrier-free Work-Stealing Definitions.

```
1: // extends definitions in Algorithm 9
2: shared variables:
3:     syncMark[c_1, ..., c_nc][2] ← [0, ..., 0][0, 0]      // one atomic integer per class, per cycle
4:     access functions:
5:         syncMark[c][cycle].get()                          // atomically read and returns value
6:         syncMark[c][cycle].set()          // atomically return and set the value, respectively
7:         syncMark[c][cycle].incGet()                       // atomically increment and returns value
```

To implement work-stealing with this strategy, no further conditions and properties other than those from the previous version are needed. The scheduler does not need any additional modification too. Hence, the scheduler algorithm is the same as the previous version.

Algorithm 13 shows the adapted worker thread with barrier-free synchronization. As we can see, this approach is quite similar to the busy-wait strategy. It replaces the usage of barriers by a matrix (defined in Algorithm 12) of atomic integers per request class per cycle to synchronize threads. Each thread involved in the execution of a request in class C must atomically increment a counter associated to C, in its respective cycle. The last thread to increment is responsible for executing the request and reset the counter. Other threads will perform stealing attempts until execution is done. The same method *Steal*() described for Algorithm 11 works also in this case.

Algorithm 13 Worker Thread for Barrier-free Synchronization

```
1: // Inherits functionality from Algorithm 11
2: variables:
3:     cycles[c_1, ..., c_nc] ← [0, ..., 0]                    // an array of cycles, one per request class
4: Worker-Thread t is as follows:
5:     while true do
6:         atomic:
7:             execQueue ← readyQueue
8:             execFlag ← readyFlag
9:             readyQueue ← ∅
10:            readyFlag ← 0
11:            readyClasses ← ∅
12:        endAtomic
13:        if execQueue ≠ ∅ then
14:            while req ← execQueue.fifoGet() do
15:                if CtoT(req.class).mode = Seq then
16:                    for all s ∈ T \ {thisTh} do
17:                        wait until marker[s] = 0
18:                    cycles[req.class] = 1 − cycles[req.class]        // current synchronization cycle for req's class
19:                    if syncMark[req.class][cycles[req.class]].incGet() = CtoT(req.class).nThreads then
20:                        exec(req)                                    // last thread to arrive executes
21:                        syncMark[req.class][cycles[req.class]].set(0)        // signal execution is done
22:                    else
23:                        while syncMark[req.class][cycles[req.class]].get() ≠ 0 do        // while exec not finished
24:                            Steal(thisTh, req.class)                 // attempt to steal, informing current class
25:                else
26:                    exec(req)
27:        else
28:            Steal(thisTh, null)                                      // attempt to steal, informing no class
```

Safety and Liveness

In this work-stealing version, both the scheduler algorithm and stealing procedure do not change. Hence, they still consistent as explained before. However, we argue that the changes in the worker thread algorithm do not invalidate its previous consistency. The atomic flags matrix used to coordinate the synchronization process was already discussed in Section 6.1. Removing the barriers structure does not affect execution flows. The new mechanism with atomic flags to control synchronization ensures the same execution flow since the arrival at execution point is corresponding to atomically increment the counter of the respective class and cycle. As it is atomic, only one thread is the last one to increment and thus responsible for execution. All other threads become free to steal. After execution, resetting the counter, in its respective class and cycle, ensures all stealer threads will continue to standard execution, as well as the victim thread.

6.2.2.3 *Choosing Victims*

The above strategy to victim selection can be improved. When attempting to steal, all free threads choose the same victim (the one with the smallest *id*), possibly resulting in contention. A better approach would be each thread to choose a different victim. Two such

strategies were implemented within this version of work-stealing. In both approaches the "Who is stolen" property was changed as described next.

a. *Random work-stealing:* whenever a thread is free to steal, it will randomly choose a thread to perform a stealing attempt. A simple modification in the steal procedure method was needed to implement this strategy;

b. *Smart work-stealing:* whenever a stealer s is free to steal, it will cyclically choose a victim v such that $v.id = s.id + 1$ to perform a stealing attempt. By cyclically we mean that when $s.id = nThreads - 1$ then it will choose victim v where $v.id = 0$. Another simple modification in the steal procedure was needed to implement this strategy.

The *random* work-stealing did not show performance gains. The *smart* approach, however, succeeded. Hence, in the next enhanced algorithm, we kept this strategy.

6.2.2.4 *Optimistic work-stealing*

We have seen so far, in this Chapter, a series of algorithms being incrementally improved with different synchronization strategies to provide better work distribution and balancing. However, the last presented algorithm, despite being wait-free while synchronizing, still has a high level of contention. When increasing the number of threads, the overhead imposed by atomic mechanisms used to provide mutual exclusion when accessing shared state (e.g. queues and synchronization controllers) can slow down system performance. This circumstance becomes evident in the experiments presented in the next Chapter, in section 7.2.2.1.

The next step of enhancement is to improve the strategy when accessing shared state to verify stealing conditions. In [25] the authors present a study of different well-established approaches of parallel algorithms. One of these approaches is called optimistic synchronization, in which, while searching for some condition in shared state, thread t does not acquire mutual exclusion objects (e. g. locks). If the condition is satisfied, then it does, and reevaluates the condition before committing execution. This strategy resorts to the idea that reevaluation succeeds most of the times, therefore called optimistic. It avoids overhead and contention by decreasing usage of mutual exclusion objects while accessing (testing) shared state only to successful situations.

Based on this concept, we implemented a version of a work-stealing algorithm that verifies stealing conditions without acquiring mutual exclusion mechanisms. Notice that such mechanisms are implicit in the atomic blocks in algorithms presented so far. The mutual exclusion is w.r.t. the victim's structures (queues, flags, *syncMark*, *readyClasses*) because they are subject to different stealers. Each stealer manipulates its own structures (*execQueue*).

Algorithm 14 The Steal Procedures for Optimistic Work-Stealing

```
 1: // Inherits functionality from Algorithm 13
 2: procedure Steal(s ∈ T, c ∈ C)
 3:     for all i ∈ [0, ..., T.length] do                          // one attempt for each thread
 4:         v ← smartPickVictim(s)                                 // choose victim using smart approach
 5:         if Validation(c, v) then                               // evaluate conditions without lock
 6:             atomic:                                            // lock() - if steal conditions satisfied, lock
 7:                 if Validation(c, v) then                       // reevaluate conditions
 8:                     s.execQueue ← v.readyQueue
 9:                     v.readyQueue ← ∅
10:                     v.marker[s] ← 1
11:                     v.readyClasses ← ∅
12:                     endAtomic                                  // unlock() after committing steal
13:                     if s.execQueue ≠ ∅ then                    // execute stolen commands
14:                         for all req in s.execQueue do
15:                             exec(req)
16:                         v.marker[s] ← 0
17:                         break for all                          // stops stealing for now
18:                 else
19:                     endAtomic                                  // unlock() - if reevaluation fails, try next victim
                                                                   // if unlocked evaluation (line 5) fails, try next victim
20: procedure Validation(c ∈ C, v ∈ T) // performs validation of stealing conditions w.r.t class c and victim v
21:     return v.readyFlag = 0 ∧
22:         v.execFlag = 0 ∧
23:         (c = null ∨
24:          NoConflict(v, c)) ∧
25:         v.readyQueue ≠ ∅
```

The optimistic work-stealing ensures the same conditions and properties as the previous versions. The scheduler does not need any additional modification. Hence, the scheduler algorithm is the same. The basic worker thread execution model does not need any additional modification too. Therefore, we present only the steal procedures.

As we can see in the Algorithm 14, the stealing algorithm chooses a victim using the *smart* approach (function *smartPickVictim*). Moreover, it verifies the stealing conditions (method *Validation*) twice. First, without acquiring mutual exclusion. If the validation succeeds, it acquires mutual exclusion and re-executes validation. If it now fails, it releases mutual exclusion without stealing, otherwise, it commits the steal and then releases mutual exclusion, proceeding normally as before.

Safety and Liveness

As stated above, the previous scheduler and worker thread algorithms do not change in this version. The only change presented here is in the process of validation of stealing conditions. The validation itself does not change. The stealing procedure execution flow has only been augmented with a pre-validation step which is executed without mutual exclusion. Yet, the real validation will take place if the former succeed, and it will be consis-

tently executed inside a mutual exclusion atomic block, ensuring the same consistency as the previous stealing procedure.

7. ENHANCEMENTS EVALUATION

We have fully implemented all algorithms described in the previous Chapter on top of the early scheduling implementation, which was built with JAVA programming language and the BFT-SMaRt state machine replication library. Now we present its evaluation.

7.1 EXPERIMENTS CONFIGURATIONS

We used the same environment, application, operation costs and workload distributions as described in experiments of Chapter 5. Configurations such as the number of server and client processes, system warm-up and duration of execution were also the same. We used equivalent class-to-threads mappings augmented to support a larger number of threads, in some cases, to provide a higher level of concurrency and opportunities to steal.

Our work-stealing technique ensures consistency by stating that only concurrent requests can be stolen, and no requests under execution by victims are conflicting with requests being stolen. Therefore not only the ratio of concurrent vs. conflicting requests matters, but also their distribution along the workload. Regarding this aspect, we experimented both uniform and exponential distribution of concurrent and sequential requests. The exponential distribution is frequently found in several phenomena such as the distribution of network packets over time, interval between requests, and others [27].

In the second case, we configured the clients to generate conflicting requests exponentially distributed among concurrent ones. We took care of keeping the same ratio between concurrent and sequential requests as in the uniform distribution. The exponential distribution generates larger chunks of concurrent requests in the workload, increasing the probability for the stealing conditions to happen. We adopted the second distribution for the purpose of measurements.

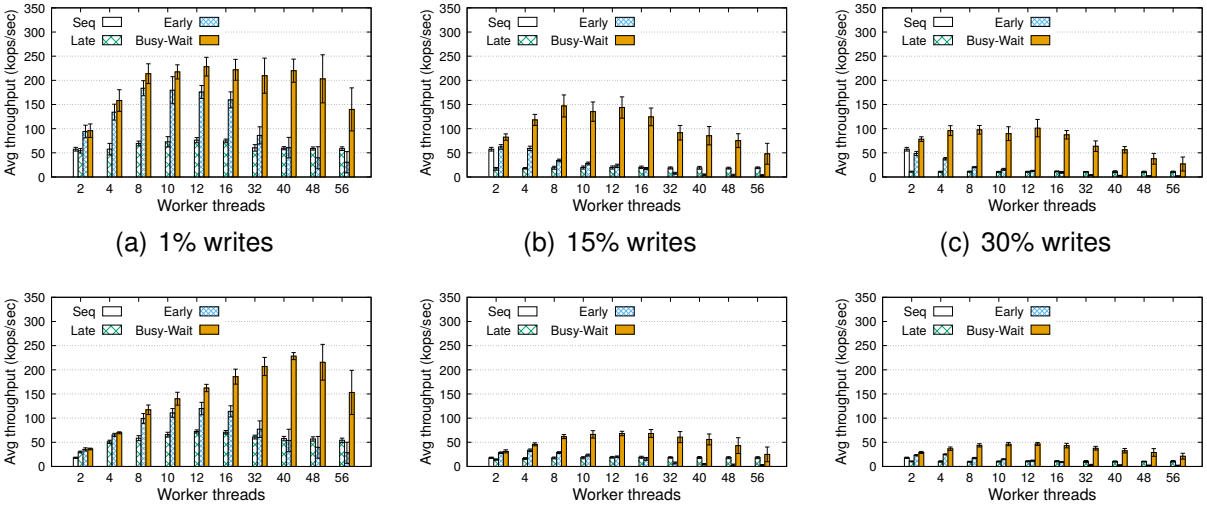
7.2 SINGLE-SHARD

We start by presenting the results for a single-sharded application experiment. In this case, client processes issue requests to the only one shard that the application in service replicas supports. We vary the percentage of conflicting requests (i.e. writes), operation costs and number of threads.

7.2.1 BUSY-WAIT RESULTS

Figure 7.1 presents the results of the busy-wait approach. As we can see, this approach shows, with 8 to 12 threads, a performance slightly better than the original early scheduling algorithm when the percentage of conflicts is low with light costs (Figure 7.1(a)). This happens because, with low levels of conflicts, there is much more concurrent execution with early scheduling, and thus much less barrier synchronization.

However, as the level of conflicts increases, we have seen in experiments of Chapter 5 that the level of idleness in the original early scheduling also increases. That happens due to higher rates of system calls caused by the blocking methods of the barriers used to synchronize threads. This phenomenon becomes evident as the busy-wait approach shows its gain, for there is no system calls to block threads, for the synchronization mechanism keeps threads active while waiting in a loop. As we can see in Figures 7.1(b) and 7.1(c), the throughput of the busy-wait approach reaches levels at about four times higher than the original early scheduling. Our busy-wait approach reaches 150k operations executed per second with 8 threads and 15% of writes. The sequential version reaches about 50k ops/sec. The late scheduling reaches about 25k ops/sec, and the early scheduling reaches at most about 55k ops/sec with 2 threads and 15% of writes. With 30% of writes, busy-wait reaches 100k ops/sec, while sequential, late and early scheduling reaches 52, 15 and 50k ops/sec, respectively.



(a) 1% writes (b) 15% writes (c) 30% writes

(d) 1% writes (e) 15% writes (f) 30% writes

Figure 7.1 – Results for single shard, light costs (top) and moderate costs (bottom).

We can also observe the impact of the operation costs in the early scheduling original approach. In Figure 7.1(d), even with a low level of conflicting requests, this approach can reach at most 110k ops/sec with 10 to 16 threads, when processing moderate costs op-

erations. On the other hand, the busy-wait reaches levels of 220k ops/sec with 40 threads. The same behavior was also observed with heavy operation costs, which we omit to avoid being repetitive. When increasing the percentage of conflicting requests, in this scenario, the performance falls for all approaches, yet the busy-wait sustains at least two times more throughput than other ones.

7.2.2 WORK-STEALING RESULTS

Although the busy-wait strategy shows performance gains, as we already stated, it is not recommended to assume an available processor for each worker thread. To hold a processing resource consuming all its capacity while waiting in a loop without doing any profitable work is not advised. Therefore we presented the work-stealing strategy, to demand stealing attempts for those misused processors, and we now evaluate such strategy.

7.2.2.1 *Work-Stealing Algorithms Evolution*

In Section 6.2 of Chapter 6 we have seen an evolution of work-stealing approaches. Each version was enhanced to afford more stealing capacity for worker threads, and thus possibly enhance the system performance.

In Figure 7.2 we can observe the comparison of each version of work-stealing and the original early scheduling approach. We represent each algorithm as follows:

- *Early:* the original early scheduling approach;
- *WS Queue:* the first work-stealing implementation, in which the steal takes place only when threads do not have work in their own queues;
- *WS Semi-b:* the second work-stealing implementation, where threads steal when waiting for synchronization too, yet proceeding to barrier synchronization when both execution or steal is done;
- *WS Non-b:* the barrier-free work-stealing approach. In this version there are no more barriers, synchronization is held by atomic variables;
- *WS Opt:* the final incremental work-stealing approach, where the stealing conditions are verified a-priori outside mutual exclusion, yet reevaluated inside it;

The incremental characteristic of each version can be seen in the results. Our two first versions, WS Queue and WS Semi-b, do not show performance gains when compared to early scheduling. Despite in some cases (2 and 4 threads in Figure 7.2(a)) they have

a throughput higher than early scheduling, in general, the three strategies present similar throughput. This result is expected, once the stealing opportunities are very small in the first version, considering that we experimented a configuration with high demand from clients, causing the queues to not be frequently empty. The second version, in turn, still has the expensive barrier synchronization.

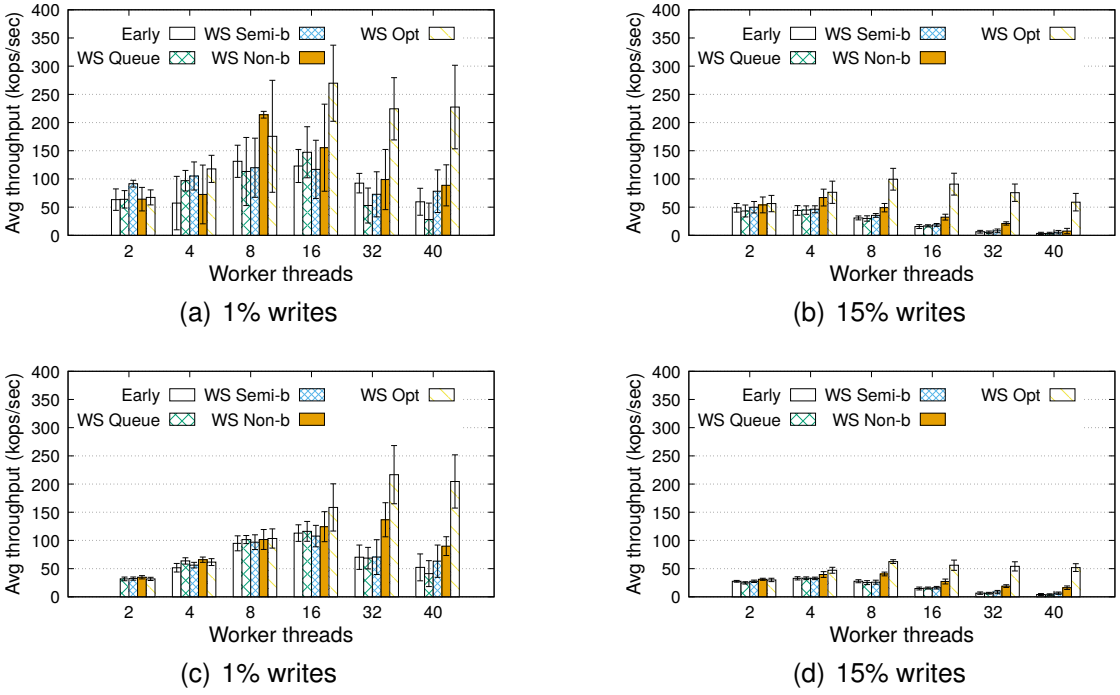


Figure 7.2 – Work-stealing algorithms comparison: single shard, light costs (top) and moderate costs (bottom).

The third work-stealing implementation (WS Non-b), in turn, presents good improvements when compared to the early scheduling technique. Removing the overhead caused by the barriers, and the contention decreased by the smart stealing strategy, it shows good results and improved the overall system performance. Finally, the optimistic (WS Opt) version surpassed all other approaches, showing a gain of more than twice the throughput of any other version, in some cases. This result was also expected, once the final version includes all improvements that were incrementally aggregated to each version. Moreover, there is a huge gain obtained when decreasing overhead by cutting usage of mutual exclusion objects while accessing shared state.

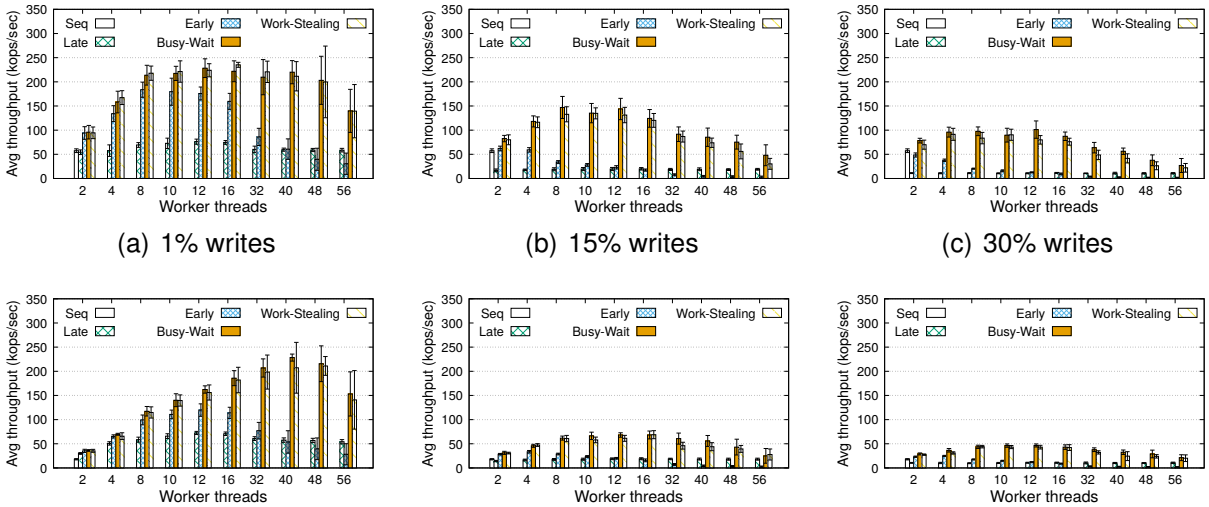
As the optimistic version has shown the best performance among all work-stealing strategies implemented, from now on we present only results of this version, when referring to work-stealing results.

7.2.3 BUSY-WAIT VS. WORK-STEALING

Here we compare the busy-wait with the final work-stealing approach that we implemented. As depicted in Figure 7.3, we found that both approaches present similar performances, statistically speaking.

Although the work-stealing mechanism affords the possibility of a better work distribution among threads, it introduces additional overhead, which ends up by restricting its performance gains. The busy-wait approach, in turn, does not introduce overhead to manage complex relations of stealing requests among threads, neither make use of expensive resources to deal with synchronization and mutual exclusion. On the contrary, it relies on simple atomic variables, which are architecture native structures, much cheaper than the structures used in work-stealing. This leads the busy-wait approach to present, in most cases considered, a performance slightly better than the work-stealing approach.

However, when limited resources are available, one would consider that a busy-wait approach is not well suitable, once it is very prone to demand high processing efforts without doing any profitable work. The work-stealing is also prone to such drawback, yet on a smaller scale, for it has much more potential to process useful information when underutilized threads succeed in stealing work from overloaded ones.



(a) 1% writes (b) 15% writes (c) 30% writes

(d) 1% writes (e) 15% writes (f) 30% writes

Figure 7.3 – Results for single shard, balanced workloads, light costs (top) and moderate costs (bottom).

Both work-stealing and busy-wait strategies reach throughput levels close to 250k ops/sec, in this single-sharded scenario, with 12 and 16 threads, light costs and low percentage of writes (Figure 7.3(a)). The behavior observed for the busy-wait approach, is also observed in work-stealing. As the level of conflicts increases, the work-stealing approach

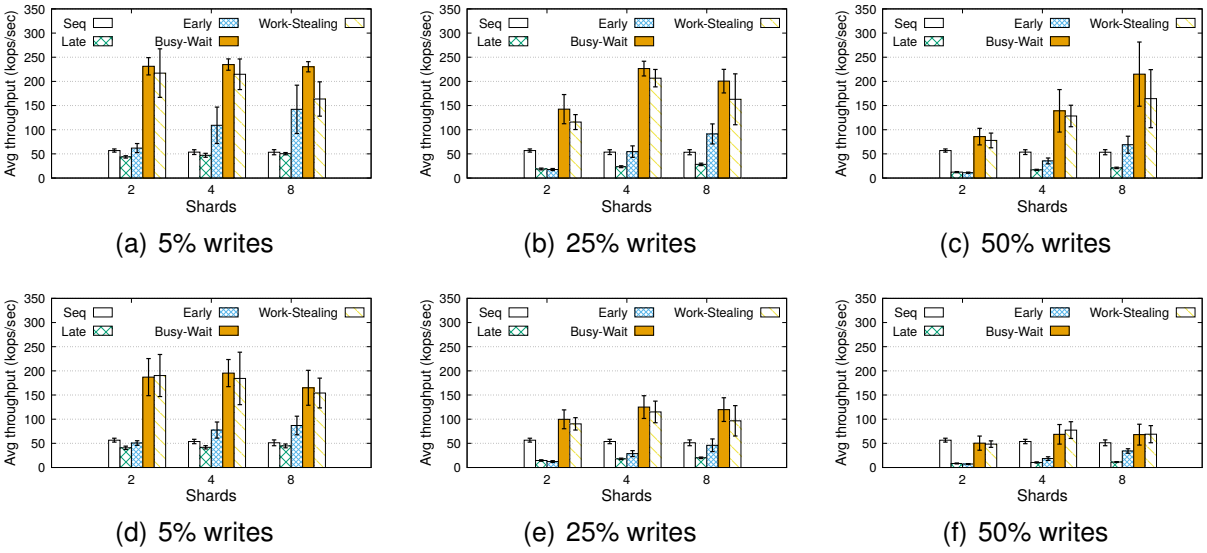
shows its gain, reaching levels at about four times higher than the original early and late scheduling algorithms (e.g. Figure 7.3(b)).

7.3 MULTI-SHARD

In this section, we present results of multi-sharded application experiments. In this case, client processes randomly choose a shard to issue requests at replicas. We considered balanced and skewed workloads, with the same distributions of requests among shards as described in the experiments of Chapter 5.

We also considered different numbers of shards, percentage of conflicting requests (i.e. writes) and percentage of global (i.e. multi-shard) operations. We fixed the number of threads to 32, and operation costs to light, based on the results from previous experiments where these configurations show the best performance.

Figures 7.4 and 7.5 present results for 2, 4 and 8 shards, with 1% and 15% of global operations issued by the clients.

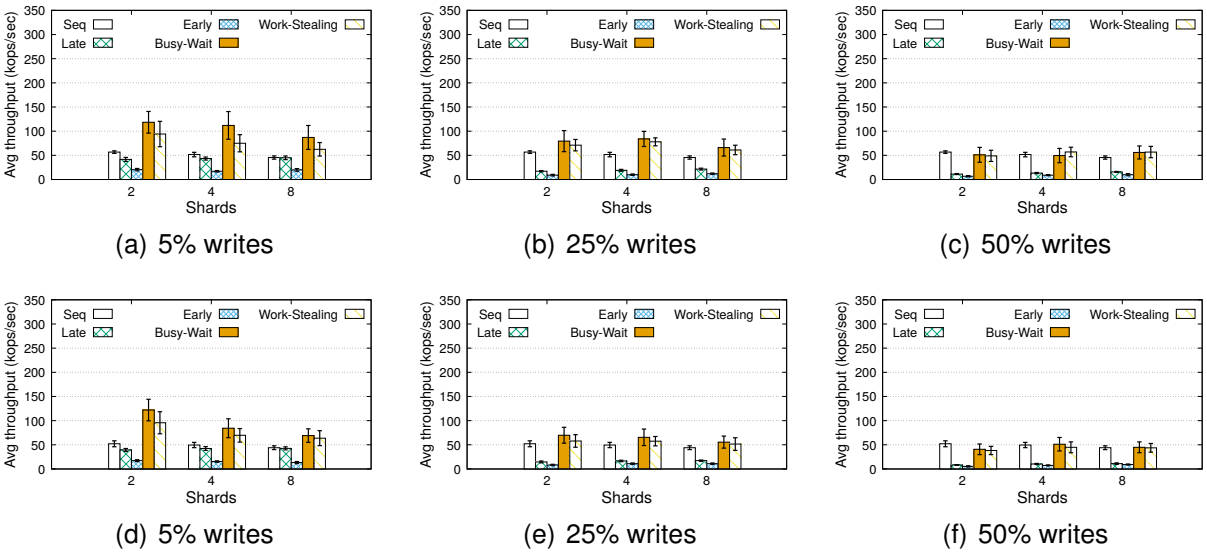


(a) 5% writes (b) 25% writes (c) 50% writes

(d) 5% writes (e) 25% writes (f) 50% writes

Figure 7.4 – Results for 1% global requests, balanced workload (top) and skewed workload (bottom).

As we can see, varying the number of shards does not have much impact on sequential and late scheduling. The same is not observed for early scheduling. Actually, it occurs the opposite: the higher number of shards, the higher concurrency of operations among shards, thus better the performance. The same occurs for busy-wait and work-stealing approaches. However, in the high conflicting balanced workload scenario (Figures 7.4(b), 7.4(c), 7.5(b) and 7.5(c)) our approaches show far better results than other ones. That happens due to the advantage of increasing both the conflicting requests and the number



(a) 5% writes (b) 25% writes (c) 50% writes

(d) 5% writes (e) 25% writes (f) 50% writes

Figure 7.5 – Results for 15% global requests, balanced workload (top) and skewed workload (bottom).

of shards. Increasing conflicts is not beneficial for the early scheduling approach (high contention and overhead caused by the barriers). But it is for our approaches, for the busy-wait does not suffer the effects of barrier overhead. Work-stealing benefits from higher opportunities of steal, either due to high level of conflicting requests and higher number of shards, for there are more opportunities of stealing concurrent requests from other shards.

Skewed workloads impact more the early scheduling and our approaches than late and sequential. However, the work-stealing approach is less impacted in more concurrent scenarios. Consider Figures 7.4(a) and 7.4(d), with 8 shards, we can see that the busy-wait approach falls from about 240k ops/sec in the balanced scenario to 160k ops/sec in skewed one. This represents a loss of 44% in throughput. The work-stealing approach falls from 160k ops/sec to about 150k ops/sec, representing only 7% of loss. Similar phenomenon occurs in Figures 7.5(a) and 7.5(d) for the case with 15% of global requests. This happens because work-stealing can minimize the skewed effect. Stealing work causes a better distribution of requests among threads in scenarios where the stealing conditions provide enough stealing capacity.

8. CONCLUSIONS

The sub-field of State Machine Replication architectures has been studied for several years in the academy, for it has great relevance in the field of distributed systems. Many efforts, as related throughout this dissertation, were done in this context to provide enhancements to SMR. Regarding parallel SMR, there are a vast number of methods to scale throughput using different approaches of scheduling decisions that allow concurrency while assuring replica consistency. Consequently, the computer industry has integrated such research results in many contexts, as already mentioned before. Given this context, the aim of our work here was to contribute with more research in this area and to the design and construction of dependable systems, especially high-throughput P-SMR. We exploited scenarios where scheduling techniques could lead to poor resource utilization. We investigated how to apply well established concurrent techniques, e.g. work-stealing, that could both reduce processor idleness and improve load balance, consequently improving system performance.

Although many works have proposed different approaches to P-SMR, we could not find any previous research integrating work-stealing concepts within P-SMR architectures. In this work we introduced this alternative, shedding some light in this not yet addressed aspect in the P-SMR literature, suggesting the study and application of work-stealing concepts in P-SMR. To do so, we extended an existing P-SMR approach to deal with work-stealing techniques. We presented our results, opening up new opportunities for further studies on the subject with the potential to generate new results to scalable reliable systems.

We investigated the early scheduling technique in more detail, and conducted studies on how to enhance its current execution model, with different strategies. The busy-wait approach, for example, is a quite simple approach that emerged while we were studying the integration of early scheduling and work-stealing. It came from the endeavor of observing how the overhead caused by the work-stealing consistency assurances could impact system performance. We found that simply removing the barriers was enough, in many scenarios, to overcome early scheduling restrictions and to provide considerable performance gains. Since that finding, we parallelized our research with both strategies and discussed their results.

8.1 FUTURE WORK

As stated above, this study opens up new opportunities for further research and improvements using the techniques here presented and discussed. We could extend the Algorithm 13, for example, to provide more stealing opportunities. This could be achieved

by updating the *execFlag* in advance, while executing its own requests, enabling the thread to become victim as soon as there is no more sequential requests in its *execQueue*.

Morevoer, in the future, we aim to deepen research with other scheduling approaches, investigating whether work-stealing techniques could be applied and succeeded with considerable performance gains. We intend to continue the research to review the state of the art and deepen the studies on existing protocols and implementations available. More specifically, with different scheduling techniques, we want to analyze those protocols from the resource utilization and load balance perspectives. Then, we will focus our efforts to understand how work-stealing could contribute in each specific case.

With early scheduling we experimented the possible integration of work-stealing to parallel scheduling at the replicas level. However, we also intend to conduct studies and experiments to verify if, due to design choices or other aspects, available implementations of scheduling techniques could restrict concurrency levels due to poor resource utilization and work distribution. Therefore, we aim to introduce a generalization study, from which we expect to understand how a generic approach of work-stealing could be conceived as a framework. Such framework would be suitable to any P-SMR scheduling technique which implements a minimum set of common characteristics, such as dependency tracking and intra-replica scheduling. From the studies and experience gained in this work and future research, a generalized work-stealing algorithm would be conceived, targeted at supporting highest possible portability through a generic integration interface.

REFERENCES

[1] Acar, U. A.; Blelloch, G. E.; Blumofe, R. D. “The data locality of work stealing”. In: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, 2000, pp. 1–12.

[2] Agrawal, K.; Fineman, J. T.; Lu, K.; Sheridan, B.; Sukha, J.; Utterback, R. “Provably good scheduling for parallel programs that use data structures through implicit batching”. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, 2014, pp. 84–95.

[3] Agrawal, K.; Leiserson, C. E.; Sukha, J. “Helper locks for fork-join parallel programming”. In: Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming, 2010, pp. 245–256.

[4] Alchieri, E.; Dotti, F.; Marandi, P.; Mendizabal, O.; Pedone, F. “Boosting state machine replication with concurrent execution”. In: Proceedings of the Eighth Latin-American Symposium on Dependable Computing, 2018, pp. 77–86.

[5] Alchieri, E.; Dotti, F.; Mendizabal, O. M.; Pedone, F. “Reconfiguring parallel state machine replication”. In: Proceedings of the IEEE 36th Symposium on Reliable Distributed Systems, 2017, pp. 104–113.

[6] Alchieri, E.; Dotti, F.; Pedone, F. “Early scheduling in parallel state machine replication”. In: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 82–94.

[7] Batista, E.; Alchieri, E.; Dotti, F.; Pedone, F. “Resource utilization analysis of early scheduling in parallel state machine replication”. In: Proceedings of the 9th Latin-American Symposium on Dependable Computing, 2019, pp. 1–10.

[8] Berenbrink, P.; Friedetzky, T.; Goldberg, L. A. “The natural work-stealing algorithm is stable”. In: Proceedings of the IEEE International Conference on Cluster Computing, 2001, pp. 178–187.

[9] Bessani, A.; Sousa, J.; Alchieri, E. E. P. “State machine replication for the masses with BFT-SMART”. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 355–362.

[10] Blelloch, G. E.; Gibbons, P. B.; Matias, Y. “Provably efficient scheduling for languages with fine-grained parallelism”. In: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, 1995, pp. 1–12.

[11] Blumofe, R. D.; Leiserson, C. E. “Scheduling multithreaded computations by work stealing”. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 356–368.

[12] Burrows, M. “The chubby lock service for loosely-coupled distributed systems”. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006, pp. 335–350.

[13] Burton, F. W.; Sleep, M. R. “Executing functional programs on a virtual tree of processors”. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, 1981, pp. 187–194.

[14] Chandra, T. D.; Toueg, S. “Unreliable failure detectors for reliable distributed systems”, *Journal of the ACM*, vol. 43–2, Mar. 1996, pp. 225–267.

[15] Cole, R.; Ramachandran, V. “Analysis of randomized work stealing with false sharing”. In: Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 985–998.

[16] Corbett, J. C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J. J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; Hsieh, W.; Kanthak, S.; Kogan, E.; Li, H.; Lloyd, A.; Melnik, S.; Mwaura, D.; Nagle, D.; Quinlan, S.; Rao, R.; Rolig, L.; Saito, Y.; Szymaniak, M.; Taylor, C.; Wang, R.; Woodford, D. “Spanner: Google’s globally-distributed database”. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, 2012, pp. 251–264.

[17] Cui, H.; Gu, R.; Liu, C.; Chen, T.; Yang, J. “Paxos made transparent”. In: Proceedings of the 25th Symposium on Operating Systems Principles, 2015, pp. 105–120.

[18] Défago, X.; Schiper, A.; Urbán, P. “Total order broadcast and multicast algorithms: Taxonomy and survey”, *ACM Computing Surveys*, vol. 36–4, Dec. 2004, pp. 372–421.

[19] Eager, D. L.; Lazowska, E. D.; Zahorjan, J. “A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)”. In: Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems, 1985, pp. 1–3.

[20] Fischer, M. J.; Lynch, N. A.; Paterson, M. S. “Impossibility of distributed consensus with one faulty process”, *Journal of the ACM*, vol. 32–2, Apr. 1985, pp. 374–382.

[21] Glendenning, L.; Beschastnikh, I.; Krishnamurthy, A.; Anderson, T. “Scalable consistency in scatter”. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 15–28.

[22] Guo, Z.; Hong, C.; Yang, M.; Zhou, D.; Zhou, L.; Zhuang, L. “Rex: Replication at the speed of multi-core”. In: Proceedings of the 9th European Conference on Computer Systems, 2014, pp. 1–14.

[23] Hadzilacos, V.; Toueg, S. “Fault-tolerant broadcasts and related problems”. In: *Distributed Systems*, Mullender, S. (Editor), New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 97–145.

[24] Halstead, Jr., R. H. “Implementation of multilisp: Lisp on a multiprocessor”. In: Proceedings of the ACM Symposium on LISP and Functional Programming, 1984, pp. 9–17.

[25] Herlihy, M.; Shavit, N. “The Art of Multiprocessor Programming, Revised Reprint”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, 536p.

[26] Herlihy, M.; Wing, J. M. “Linearizability: A correctness condition for concurrent objects”, *ACM Transactions on Programing Languages and Systems*, vol. 12–3, Jul. 1990, pp. 463–492.

[27] Jain, R. “The Art Of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling”. Noida, Uttar Pradesh, India: Wiley India Pvt. Limited, 2008, 685p.

[28] Kapitza, R.; Schunter, M.; Cachin, C.; Stengel, K.; Distler, T. “Storyboard: Optimistic deterministic multithreading”. In: Proceedings of the Sixth International Conference on Hot Topics in System Dependability, 2010, pp. 1–8.

[29] Kapritsos, M.; Wang, Y.; Quema, V.; Clement, A.; Alvisi, L.; Dahlin, M. “All about Eve: Execute-verify replication for multi-core servers”. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012, pp. 237–250.

[30] Kotla, R.; Dahlin, M. “High throughput byzantine fault tolerance”. In: Proceedings of the International Conference on Dependable Systems and Networks, 2004, pp. 575–584.

[31] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, vol. 21–7, Jul. 1978, pp. 558–565.

[32] Leung, J.; Kelly, L.; Anderson, J. H. “Handbook of Scheduling: Algorithms, Models, and Performance Analysis”. Boca Raton, FL, USA: CRC Press, Inc., 2004, 1224p.

[33] Marandi, P. J.; Bezerra, C. E.; Pedone, F. “Rethinking state-machine replication for parallelism”. In: Proceedings of the IEEE 34th International Conference on Distributed Computing Systems, 2014, pp. 368–377.

[34] Mendizabal, O. M.; Moura, R. S. T. D.; Dotti, F. L.; Pedone, F. “Efficient and deterministic scheduling for parallel state machine replication”. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2017, pp. 748–757.

[35] Olszewski, M.; Ansel, J.; Amarasinghe, S. “Kendo: efficient deterministic multithreading in software”, *ACM Sigplan Notices*, vol. 44–3, Mar 2009, pp. 97–108.

[36] Perez, J. M.; Badia, R. M.; Labarta, J. “A dependency-aware task-based programming environment for multi-core architectures”. In: Proceedings of the IEEE International Conference on Cluster Computing, 2008, pp. 142–151.

[37] Rossbach, C. J.; Currey, J.; Silberstein, M.; Ray, B.; Witchel, E. “Ptask: Operating system abstractions to manage gpus as compute devices”. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 233–248.

[38] Schneider, F. B. “Implementing fault-tolerant services using the state machine approach: A tutorial”, *ACM Computing Surveys*, vol. 22–4, Dec. 1990, pp. 299–319.

[39] Squillante, M. S.; Nelson, R. D. “Analysis of task migration in shared-memory multiprocessor scheduling”. In: Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems, 1991, pp. 143–155.

[40] Tchiboukdjian, M.; Gast, N.; Trystram, D. “Decentralized list scheduling”, *Annals of Operations Research*, vol. 207, Aug 2013, pp. 237–259.

[41] Yang, J.; He, Q. “Scheduling parallel computations by work stealing: A survey”, *International Journal of Parallel Programming volume*, vol. 46–2, Apr 2018, pp. 173–197.

[42] Yao, X.; Geng, P.; Du, X. “A task scheduling algorithm for multi-core processors”. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies, 2013, pp. 259–264.